Caracterización de Aplicaciones de Paso de Mensajes con Técnicas de Caja Negra

Tesista

Eduardo Grosclaude

Directores

Ing. Armando De Giusti

Dr. Marcelo Naiouf

Trabajo Final presentado para obtener el grado de
Especialista en Cómputo de Altas Prestaciones y Tecnología Grid

Facultad de Informática
Universidad Nacional de La Plata
Marzo 2012

Caracterización de Aplicaciones de Paso de Mensajes con Técnicas de Caja Negra

Eduardo Grosclaude

6 de marzo de 2012

Resumen

La mayor parte de la literatura sobre análisis de rendimiento de programas paralelos está orientada a la optimización a cargo del programador. Por lo tanto, supone conocimiento de los algoritmos empleados, su orden de complejidad, su forma de implementación, y en definitiva, asume acceso a los fuentes de esos programas. El instrumental disponible de análisis de rendimiento se basa mayormente, entonces, en propiedades estructurales de los programas, que sólo se hacen evidentes a partir del código fuente. Sin embargo, en ocasiones el analista de rendimiento en Cómputo de Altas Prestaciones se enfrenta al problema de optimizar la ejecución de aplicaciones en cuyo desarrollo no ha intervenido, y de las cuales sólo se tiene la versión final ejecutable. Otras veces se dispone de los fuentes, pero su interpretación es de todos modos costosa, debido a la complejidad de estudiar código escrito por terceros, o debido a la dificultad de comprender el dominio del problema. Si se dispone únicamente de los programas en su versión binaria, las técnicas de predicción de prestaciones que se apoyan en el conocimiento de los fuentes no pueden ser aplicadas, y el analista debe normalmente proceder por ensayo y error. El trabajo que se propone intenta identificar una metodología conveniente para obtener conocimiento acerca de las aplicaciones, en ausencia del código fuente, que permita hacer inferencias sobre su rendimiento en diferentes plataformas, existentes o hipotéticas, con el fin de seleccionar las mejores plataformas para cada aplicación.

Índice

1. Introducción

El Proyecto de Investigación 04/E085, “Cómputo de Altas Prestaciones”, de la Facultad de Informática de la Universidad Nacional del Comahue, tiene el objetivo general de "adquirir conocimientos para el diseño, desarrollo, gestión y mejora de las tecnologías de hardware y software involucradas en la Computación de Altas Prestaciones (CAP) y sus aplicaciones en Ciencia e Ingeniería Computacional" [2]. El proyecto fue iniciado en 2010 y su actividad ha sido orientada tanto al estudio de aspectos teóricos como hacia la adquisición de conocimiento práctico.

Desde antes de la conformación del proyecto, sus integrantes vienen realizando cursos de postgrado sobre el tema, primero durante la vigencia del programa CYTED, y luego como parte del programa de formación de recursos humanos del proyecto de investigación. El autor de este trabajo es alumno de la carrera de

postgrado Especialista en Cómputo de Altas Prestaciones y Tecnología Grid, de la Universidad Nacional de La Plata.

1.1. Motivación de este trabajo

A los fines del proyecto ha sido de especial utilidad la relación informal de asesoramiento conformada con el Proyecto 04/I157, “Propiedades Estructurales, Electrónicas y Termodinámicas de Moléculas, Nanoestructuras y Sólidos Cristalinos”, de la Facultad de Ingeniería de la UNC. A través de esta relación se ha podido tomar conocimiento de diversos problemas originados en el ámbito real de trabajo de un grupo de investigación en ciencias computacionales.

El proyecto "Propiedades Estructurales...", como parte de su labor de investigación, ejecuta regularmente varias aplicaciones de simulación atomística, de preferencia en clusters y equipos multicore. Algunas de estas aplicaciones son SIESTA, LAMMPS, WIEN2K, ABINIT. Todas estas aplicaciones, conocidas por la comunidad de investigadores en Física, son capaces de utilizar esquemas de paralelismo de memoria distribuida haciendo uso de alguna implementación del estándar MPI; y algunas de ellas pueden funcionar además sobre máquinas paralelas de memoria compartida (con diversas implementaciones, a saber, OpenMP, MKL, u otras).

Con anterioridad a la aparición de las plataformas multicore, aplicaciones como las mencionadas eran ejecutadas en pequeños clusters de equipos multipropósito, dedicados o no. Estos equipos multipropósito eran cíclicamente renovados, y durante los años en que cada generación de hardware mejoraba la performance por el simple expediente de aumentar la frecuencia de reloj y la tecnología de memoria, la elección de nuevo hardware a la hora de actualizar el cluster era simple y directa.

Sin embargo, las nuevas estrategias de diseño de los equipos de cómputo obligan a la reflexión sobre qué alternativas elegir. Diferentes plataformas tienen diferente organización y potencialidades, y una mala decisión de inversión puede desaprovechar capital o tiempo del proyecto. Consecuentemente, entre las demandas de asesoramiento planteadas por dicho proyecto se encuentra la concerniente a cómo dimensionar adecuadamente el equipamiento para la ejecución de sus programas.

Dada la variada oferta actual en hardware, se reconoce que no es ésta una problemática menor. Entre las ofertas recientes se encuentra una amplia gama de configuraciones, contando con desde dos hasta ocho núcleos. A éstas se deben agregar las potenciales ofertas de hardware con aún mayor cantidad de recursos, previsibles a futuro cercano. Estas opciones pueden a su vez configurarse en un cluster híbrido en muchas posibles combinaciones, lo cual aumenta la incertidumbre del usuario no especialista a la hora de invertir en equipamiento.

Para poder ofrecer una respuesta racional a esta pregunta, es necesario contar con conocimiento de las aplicaciones que van a ser ejecutadas, para poder mapear sus requerimientos a los recursos de las plataformas disponibles. De aquí surge el problema principal que trata este trabajo, ya que en general, las aplicaciones utilizadas por el grupo son diseñadas y programadas por terceros. En estas condiciones, es difícil o costoso conocer detalles de implementación tales como los necesarios para caracterizar el mejor hardware para una aplicación, ya que:

- Muchas de las aplicaciones usadas por el grupo son cerradas, es decir, no se dispone de los fuentes;
- En los casos en que se dispone de los fuentes, se necesita conocimiento del dominio del problema para poder interpretarlos, conocimiento que es costoso de adquirir.

Dado todo lo anterior, para poder aproximarse a las respuestas buscadas, el proyecto Cómputo de Altas Prestaciones ha buscado configurar una metodología para el análisis de prestaciones de programas paralelos a caja negra, es decir, sin apoyo de conocimiento a priori alguno sobre las aplicaciones. La intención de la metodología es poder predecir prestaciones sobre diferentes plataformas, hipotéticas o reales, de manera de permitir la comparación y orientar decisiones.

En este trabajo, con el fin de satisfacer los requisitos para el título de Especialista en Cómputo de Altas Prestaciones, se expone el desarrollo y aplicaciones de dicha metodología. La mayor parte de las ideas y experiencias que conforman este trabajo han sido publicadas en forma de artículos presentados en encuentros nacionales JAIIO (HPC), CACIC (WPDP) y WICC durante 2010 y 2011 [29, 30, 32, 31].

1.2. Plan del trabajo

En las secciones siguientes se revisarán principios de análisis de prestaciones de aplicaciones paralelas; se comentarán las formas de instrumentación de aplicaciones de paso de mensajes, y se hará un relevamiento de herramientas existentes de análisis de prestaciones. Se describirán las formas corrientes de instrumentación de aplicaciones utilizando MPI, y su relación con el uso de bibliotecas de carga dinámica. Teniendo en cuenta la técnica de instrumentación por intercepción de funciones elegida, se describirá la arquitectura y diseño del framework de instrumentación propuesto, y se darán ejemplos de aplicación.

2. Análisis de prestaciones

Tanto la estructura estática como el comportamiento dinámico son información valiosa para la optimización de programas. Mientras que la información estructural es conocida por el programador, antes de la ejecución, la información dinámica de los programas puede ser recogida por terceros, durante la ejecución. Diferentes clases de información sugieren diferentes estrategias para la optimización.

La mayor parte de la literatura sobre análisis de rendimiento de programas paralelos está orientada a la optimización a cargo del programador. Por lo tanto, supone conocimiento de los algoritmos empleados, su orden de complejidad, su forma de implementación, y en definitiva, asume acceso a los fuentes de esos programas. El instrumental disponible de análisis de rendimiento se basa mayormente, entonces, en propiedades estructurales de los programas, que sólo se hacen evidentes a partir del código fuente.

Sin embargo, en ocasiones el analista de rendimiento en Cómputo de Altas Prestaciones se enfrenta al problema de optimizar la ejecución de aplicaciones en cuyo desarrollo no ha intervenido, y de las cuales sólo se tiene la versión final ejecutable. En este escenario cambian, tanto el conocimiento que es posible obtener sobre el programa, como la clase de acciones que pueden tomarse para, a partir de ese conocimiento, modificar una situación dada. Mientras que el programador con acceso a los fuentes puede modificar las aplicaciones para mejorar las prestaciones, en el caso opuesto se pueden determinar estrategias de scheduling, de reubicación o migración de procesos en diferentes procesadores, o en general seleccionar diferentes máquinas paralelas donde ejecutar la aplicación.

2.1. Instrumentación de código

La instrumentación de programas es la inserción de instrucciones en el código, con el fin de registrar información sobre el comportamiento dinámico. Esta información dinámica puede referirse a cualquier conjunto de eventos interesantes, y ser presentada en varias formas: particularizada evento por evento (*tracing*), o agregada (*profiling*). Cuando la estructura del programa es desconocida, la instrumentación del código puede resultar una forma de conocer detalles del comportamiento, y aun de reconstruir una vista parcial de la información estática del programa.

La optimización racional de las aplicaciones requiere que la inserción de código extraño tenga un impacto mínimo sobre la ejecución, y sobre todo, que la salida del programa no se vea modificada.

2.2. Clases de instrumentación

La instrumentación de un programa puede ser efectuada por dispositivos automáticos o por un programador, y puede ocurrir a varios niveles:

Instrumentación de código fuente. Al nivel de los fuentes, el programador puede modificar el código para incluir instrucciones de registro de eventos. Este método es el más flexible, pero también laborioso; requiere recompilación, puede colisionar con optimizaciones efectuadas por el compilador, y el producto final normalmente no es el apto para producción debido a la sobrecarga introducida por la instrumentación. Esta instrumentación se efectiviza al tiempo de compilación.

Instrumentación de código objeto. Usando bibliotecas *wrapper*, los programas pueden revelar información sobre las llamadas a funciones que realizan. Las llamadas son interceptadas por herramientas de análisis de performance que reimplementan las funciones de biblioteca, y son capaces de contabilizar estos eventos. Sin embargo, mientras el control se encuentra dentro de dichas funciones reimplementadas, puede no ser posible recuperar la información que permita correlacionar el evento con las líneas fuente correspondientes, dificultando el análisis. La instrumentación por *wrappers* normalmente funciona revinculando los módulos objeto de la aplicación con las nuevas bibliotecas, y por lo tanto se produce al tiempo de linkedición.

Instrumentación de nivel binario. Es la inserción de código de registro de eventos directamente en el archivo ejecutable o imagen del programa en memoria. Tanto las técnicas de instrumentación de nivel fuente como las de nivel de objeto requieren algún grado de acceso a componentes separados de la aplicación, ya sea para editar los fuentes como para revincular los módulos objeto y reconstruir un ejecutable. En cambio, la instrumentación binaria puede ofrecer una técnica de análisis de performance de caja negra, prescindiendo de los fuentes y contando únicamente con el ejecutable en su forma final. Sin embargo, al nivel del análisis binario, los eventos no pueden correlacionarse con las instrucciones fuente originales. Aunque de implementación cómoda para el usuario, esta clase de instrumentación es limitada y técnicamente compleja. Requiere un conocimiento profundo de la arquitectura subyacente y no es fácilmente portable. La instrumentación binaria puede ser una herramienta poderosa para la sintonización dinámica [34, 39]. Tiene lugar al tiempo de ejecución.

2.3. Trazas y perfiles

Los productos típicos de la instrumentación de código son trazas y perfiles.

- Una traza de un programa es una cadena de registros de eventos de ejecución, generalmente junto con sus tiempos de ocurrencia o *timestamps* y posiblemente algunas observaciones de datos o métricas asociadas.
- Un perfil de programa es la información cuantitativa resumida sobre un conjunto de eventos verificados durante una ejecución.

Mientras que las trazas sirven para describir la vida completa de una ejecución, los perfiles usualmente consideran métricas orientadas a tiempos o número de eventos registrados. Los eventos de interés para las trazas o perfiles de programas pueden estar relacionados con actividad de nivel de máquina, llamadas a funciones internas del programa, o llamadas a funciones de bibliotecas externas.

Eventos de máquina. Las arquitecturas modernas proveen contadores de hardware, que contabilizan determinados eventos de máquina como cantidad de instrucciones ejecutadas, *hits* o *misses* de *cache* o de TLB, resultados de predicciones de saltos, u operaciones de punto flotante. Cuando estos contadores entran en condición de *overflow*, o al alcanzar algún umbral prefijado, generan interrupciones. Los usuarios pueden atrapar estas interrupciones, con una mínima intrusión en las aplicaciones que se están ejecutando, para efectuar cualquier tarea relacionada (como incorporar información a una traza, o contabilizar eventos con fines de perfilado). La principal dificultad de esta técnica, debido a su bajo nivel, consiste en lograr la correcta atribución de los eventos a los procesos de interés, discriminando los eventos que corresponden a la actividad del sistema operativo u otras tareas concurrentes en el mismo sistema.

Llamadas a funciones. Una aplicación puede perfilar el uso de sus propias funciones para revelar dónde transcurre cada porción de su tiempo de ejecución. Así, el programador que desea optimizar un programa manipulando los fuentes, puede obtener el mayor provecho de sus esfuerzos dedicándose a las regiones del programa que consumen más tiempo de ejecución.

Llamadas a bibliotecas externas. Frecuentemente, los programadores de aplicaciones paralelas tienen interés en otras llamadas, como las relacionadas con cómputo especial o comunicaciones. Conocer la cantidad de funciones de comunicación que son invocadas, y cuánto tiempo demoran, puede revelar conocimiento sobre la escalabilidad o los cuellos de botella en la aplicación. Las funciones de biblioteca MPI son candidatas naturales para el trazado y perfilado.

Aunque el trazado provee información detallada y específica, las trazas pueden ser conjuntos de datos de muy gran tamaño y costosas de procesar, requiriéndose estructuras de datos y algoritmos especiales [19]. Las mejores herramientas

de análisis de performance para manipular trazas son aquellas que permiten al usuario visualizarlas y realizar interactivamente sus consultas.

Por otra parte, los perfiles son piezas densas de información que pueden señalar rápidamente cuellos de botella o zonas de programa donde la optimización puede causar mayores efectos, pero carecen de detalle y no proveen una visión de la ejecución del programa a lo largo del tiempo. Los perfiles se prestan mejor a la comparación de estadísticos entre grupos de corridas o experimentos.

Algunos abordajes híbridos intentan preservar la información relacionada con el tiempo, a la vez manteniendo el tamaño de los datos dentro de límites manejables (como en el perfilado incremental [26, 40], donde la conducta de la aplicación se describe por una secuencia de instantáneas).

2.4. Herramientas de Análisis de Prestaciones

Actualmente están disponibles numerosas herramientas de análisis de prestaciones, que operan mediante trazado o perfilado, con diversas capacidades. Se han publicado varios relevamientos o rankings de herramientas, con acuerdo a varios criterios [20, 39, 42, 11]. Aunque la portabilidad es un criterio frecuentemente utilizado, aquí nos referiremos a algunas herramientas en el dominio de las aplicaciones para Linux.

Herramientas orientadas a máquina. Existen varias herramientas que explotan las capacidades de monitorización del hardware. Herramientas como OPROFILE [1] actúan en forma independiente, mientras que PAPI [22], PERFSUITE [6], TAU [14], dependen de servicios provistos por capas inferiores tales como los implementados por PERFCTR [13].

Aunque existen desde hace bastante tiempo bibliotecas de instrumentación de bajo nivel, la capacidad del kernel Linux de utilizar contadores de hardware recién aparece en versiones recientes, primero con el nombre de PCL (PERFORMANCE COUNTERS FOR LINUX) [21], y posteriormente rebautizado PERF EVENTS. PCL es un subsistema que busca sustituir a oprofile, cuyo uso presenta una cantidad de inconvenientes. La herramienta de usuario PERF, diseñada para el proyecto PCL, provee contadores, agrupaciones de contadores y dispositivos de muestreo, discriminando los alcances global, por tarea y por CPU. Los eventos de sistema operativo, como fallas de página menores y mayores, migración de tareas entre procesadores, o cambios de contexto, también son registrados.

STRACE y GPROF. Strace es un programa de trazado de procesos conocido por los usuarios de los modernos sistemas de la familia UNIX. Durante una corrida bajo control de strace, se evidencian las llamadas al sistema emitidas

por el programa junto con sus timestamps, argumentos y resultados. Varias opciones de strace permiten refinar las consultas, pudiéndose definir una familia de llamadas al sistema, o computar el tiempo transcurrido durante cada llamada. GNU gprof [4] es otra herramienta familiar en el universo UNIX. Luego de la ejecución (*post-mortem*) de una aplicación compilada con soporte de gprof, la herramienta gprof emite un árbol de llamadas, ordenando las funciones del programa por número de llamadas o tiempo insumido.
Gprof puede funcionar con aplicaciones MPI. Cuando se define la variable de ambiente GMON_OUT_PREFIX, cada proceso participante en una corrida MPI escribe en un archivo, con un nombre privado, un perfil que describe estadísticamente su ejecución. Los perfiles marginales pueden luego combinarse o resumirse mediante gprof, presentando una única vista global de la corrida.

MPE, Jumpshot. MPE (Multi Processing Environment) viene con la implementación MPICH de MPI [7], pero puede ser usado con otras implementaciones MPI con algo de adaptación. MPE es probablemente el *framework* de análisis de prestaciones para MPI más popular a causa de la extensa base de usuarios de MPICH, y de la herramienta visual libremente disponible JUMPSHOT. MPE es una biblioteca de tiempo de vinculación, pero también provee un conjunto de funciones que puede usar el programador para desarrollar sus propias estrategias de trazado de programas. Jumpshot es una aplicación Java capaz de trabajar en forma efectiva con trazas de gran tamaño. El formato de traza en MPE ha sufrido bastantes cambios evolutivos (ALOG, BLOG, CLOG, SLOG, SLOG2...) para enfrentar los problemas de escalabilidad que han ido surgiendo.

VAMPIRTRACE y VAMPIR. VampirTrace es un dispositivo de trazado que viene con la biblioteca Open MPI [27, 28], implementado como un *wrapper* al compilador MPICC más una biblioteca de manipulación de trazas LIBOTF. Durante la ejecución, una aplicación compilada con soporte de VampirTrace genera un conjunto de archivos incluyendo un archivo de traza por proceso. El formato para estos archivos se conoce como Open Trace Format (OTF) [15].
El *framework* viene con algunos utilitarios simples para explotar las trazas OTF. Sin embargo, la única interfaz gráfica, interactiva, para manipular trazas OTF que hemos podido identificar es Vampir, un producto comercial con versiones de evaluación [17]. Por otro lado, existe una carencia notable en materia de herramientas de conversión desde o hacia otros formatos de traza [38]. Estos son hechos más bien sorprendentes, ya que la especifi-

cación del formato de trazas y biblioteca están publicadas como software Open Source bajo una licencia no restrictiva. A diferencia de MPE y Jumpshot, que son herramientas confinadas a conceptos MPI, VampirTrace es más general, ya que provee un mecanismo para registrar y analizar no solamente eventos MPI sino también aquellos en dominios diferentes, como los eventos de máquina o los de threads OpenMP.

mpiP. Propuesta como una biblioteca de perfilado MPI liviana y escalable [9, 46], mpiP se vincula con un archivo objeto, y al tiempo de ejecución intercepta las llamadas a MPI dentro de un proceso. Los resultados para todos los procesos se combinan al final de la corrida, de modo que no se requiere comunicación extra entre procesos para obtener la vista global.
La salida de mpiP, legible por humanos, identifica las llamadas a MPI en el programa y resume la cantidad de llamadas y tráfico de datos para cada punto del programa (*callsite*) en varias maneras. Cuando los fuentes se compilan con soporte de depuración, mpiP puede correlacionar las estadísticas de cada *callsite* con las líneas fuente. Los fuentes de la biblioteca mpiP son libremente accesibles.

3. Instrumentación de aplicaciones a caja negra

3.1. Interfaz de perfilado de MPI

La definición de MPI [8] incluye un mecanismo para facilitar y estandarizar el trabajo de los desarrolladores de herramientas de perfilado y otras similares, condensado en la definición de la Interfaz de Perfilado (MPI PROFILING INTERFACE LAYER). Para satisfacer los requerimientos de la definición del estándar, las implementaciones válidas de MPI deben contener un punto de entrada secundario a las rutinas de la biblioteca.

Desde la vista del programador de aplicaciones, todas las funciones MPI de cualquier implementación tienen nombres con el prefijo *MPI_*. La Interfaz de Perfilado asegura que la implementación también contendrá un conjunto equivalente de nombres para las mismas funciones, los cuales se corresponden, uno a uno, pero se distinguen por el prefijo *PMPI_*. La interfaz no establece cómo debe ejecutarse el perfilado, ni se limita a ese propósito, sino que consiste en un mecanismo general para la construcción de herramientas que deban sincronizarse con las llamadas a funciones de biblioteca, para fines cualesquiera.

Con este mecanismo, las herramientas podrán implementar bibliotecas cuyas funciones tengan nombres superpuestos a los prefijados con *MPI_*, pero además podrán utilizar los nombres con prefijo *PMPI_* para hacer sus propios accesos

a la biblioteca MPI. Una biblioteca de perfilado puede superponerse total o parcialmente (es decir, implementando todas o algunas de las funciones MPI). Al vincular los módulos objeto de una aplicación contra la biblioteca de una herramienta de perfilado, la conducta del *linker* para cada símbolo *MPI_* pendiente de resolución dependerá de si la función está o no implementada en la biblioteca de perfilado. Si está implementada, el *linker* resolverá las colisiones entre los nombres ligando el código a la función de la biblioteca de perfilado; si no lo está, la vinculación se efectuará con la implementación en la biblioteca MPI original. Esta lógica se impone simplemente por el orden de aparición entre los parámetros dados al linker al tiempo de vinculación. La existencia de las funciones *PMPI_* permite evitar las ambigüedades o dependencias cíclicas.

Al tiempo de ejecución, las llamadas a funciones *MPI_* de la aplicación serán interceptadas por aquellas implementadas por la herramienta, en forma funcionalmente transparente, aunque al precio de algún *overhead* esperablemente pequeño. Como el diseño de la interfaz de perfilado pretende ser portable respecto de implementaciones y plataformas, los usuarios no necesitan tener acceso al código o detalles de implementación de MPI. Sin embargo, normalmente el usuario debe, al menos, revincular su aplicación contra la biblioteca de perfilado que utiliza la interfaz, y para esto, según el procedimiento usual de compilación y vinculación, los módulos objeto de la aplicación deben ser accesibles a los usuarios no propietarios. Como este requerimiento va en contra de nuestra premisa de prescindir de otros elementos que los binarios ejecutables en su forma final, recurriremos a otro modo de vinculación.

3.2. Vinculación dinámica

Dynamic Shared Objects (DSO)

A partir de la aparición de los mecanismos de memoria virtual en los sistemas operativos, ha sido posible disminuir el tamaño en disco y en memoria de los programas y aumentar la capacidad de multiprogramación, con el uso de bibliotecas de código compartido. Los primitivos formatos binarios de bibliotecas estáticas de UNIX, correspondientes a los ejecutables de los tipos *a.out* o COFF, no contemplaban la relocalización de código, por lo cual efectivamente impedían el uso de código compartido. Para superar este y otros problemas técnicos, los modernos sistemas de la familia UNIX adoptaron el formato de binarios ELF (EXECUTABLE LINKAGE FORMAT) [3] con lo cual la construcción de bibliotecas compartidas (DSO, o DYNAMIC SHARED OBJECTS) se facilitó enormemente. Los DSOs son básicamente programas ELF sin dirección de carga fija [24].

Los modernos sistemas de desarrollo compilan los programas, por defecto, en modo de vinculación dinámica (es decir, como usuarios de DSOs), lo que implica

que los textos de las funciones de bibliotecas referenciadas no integran el módulo objeto sino que las referencias quedan pendientes hasta el momento de carga. Al ingresar del almacenamiento a la memoria, los programas dinámicamente vinculados no quedan completos, en disposición de ser ejecutados; sino que, antes de que se pueda transferir el control al nuevo proceso, el vinculador dinámico (*dynamic linker*) deberá completar la aplicación usando los DSOs presentados al linker al momento de linkedición.

El vinculador dinámico deberá:

1. Determinar las dependencias, construyendo una lista de búsqueda (*lookup scope*) formada por los DSOs necesarios, y cargarlas.

2. Relocalizar la aplicación y todas sus dependencias:

   a) Las que son conocidas y presentes en el mismo módulo objeto, identificadas por la posición relativa de la ubicación dentro del objeto.

   b) Las basadas en símbolos, cuya definición está generalmente en un módulo objeto diferente al que la referencia.

3. Inicializar la aplicación y las dependencias en el orden correcto.

La fase 2.b) de este proceso requiere que el vinculador dinámico realice una búsqueda para cada símbolo a relocalizar, en cada DSO en la lista de búsqueda. La búsqueda terminará exitosamente al encontrar la primera instancia coincidente en cualquiera de los DSOs en la lista de dependencias (o fallará haciendo imposible la ejecución, si no se encuentra en ninguno de ellos).

Preloading

El vinculador dinámico ofrece una funcionalidad especial de *preloading*, prevista en la definición del formato ELF, que permite afectar la forma como se construye la lista de búsqueda *lookup scope*. Si el usuario modifica el valor de la variable de ambiente LD_PRELOAD en el contexto de un proceso que carga un programa de vinculación dinámica, indicando nombres de archivos DSO, o si especifica esos valores en un archivo de configuración (en Linux, habitualmente */etc/ld.so.preload*), entonces los DSOs mencionados aparecerán a la cabeza de la lista de búsqueda en la fase 1 del procedimiento de carga delineado anteriormente. La consecuencia es que más tarde, en la fase 2.b), dichos módulos objeto compartidos serán inspeccionados antes que las demás bibliotecas, y las referencias pendientes en la aplicación se resolverán contra los símbolos definidos en los DSOs especificados.

Vinculación dinámica explícita

La funcionalidad de *preloading* se complementa con la API POSIX de carga dinámica (*dynamic linking*) explícita de DSOs. Esta API provee funciones para que las aplicaciones puedan utilizar DSOs explícitamente al tiempo de ejecución, una vez superada la instancia de carga y relocalización de dependencias. El mecanismo permite la vinculación tardía de objetos, aun aquellos cuya definición se encuentra en bibliotecas aún no determinadas al tiempo de carga.

La API de carga dinámica contiene funciones para abrir un DSO (función *dlopen()*), obteniendo un *handle* para futuras operaciones, y para obtener la dirección de un símbolo allí definido (función *dlsym()*). Una aplicación abre un DSO, dado por su nombre de archivo, lo que implica efectivamente vincular todos los símbolos de variables contenidas en el DSO en el espacio del programa en ejecución. En general lo mismo ocurre con las funciones, salvo que si en la operación de apertura se especificó la opción RTLD_LAZY, las funciones contenidas en el DSO son cargadas a demanda, es decir, únicamente cuando se ejecuta el código que las referencia.

Las funciones cuyas direcciones son obtenidas mediante *dlsym()* pueden ser invocadas desde el programa por dereferenciación. La carga de DSOs incluye automáticamente a todas las dependencias del DSO abierto [16].

3.3. Intercepción de funciones

Combinando *preloading* con vinculación dinámica explícita, es posible interceptar o sustituir llamadas a funciones en una aplicación de vinculación dinámica, sin otro insumo que el archivo binario de la aplicación. La técnica combinada puede explotarse a los fines de la instrumentación en varias formas:

1. Cargando una biblioteca de perfilado de nivel de objeto, al tiempo de ejecución de una aplicación de vinculado dinámico arbitraria. Este caso de uso libera al usuario del requerimiento de contar con los módulos objeto componentes de la aplicación al tiempo de vinculación (Figura 1a).

2. El usuario puede escribir sus propias bibliotecas de perfilado, e instrumentar aplicaciones arbitrarias al tiempo de ejecución. Este caso de uso permite al usuario hacer su propio perfilado de aplicaciones cerradas, en un escenario de caja negra (Figura 1b).

3. Interponiendo las funciones del usuario contenidas en un DSO, al tiempo de ejecución, entre una aplicación y una biblioteca de perfilado hecha por terceros. Este caso de uso tiene el valor agregado de permitirnos operar

concurrentemente con otras herramientas de perfilado implementadas como bibliotecas compartidas, que exploten MPI Profiling Interface Layer, y obtener los resultados independientes de cada herramienta para una misma corrida (Figura 1c).

4. El caso de uso 3 es iterable. El *lookup scope* puede definirse de modo de hacer referencia a una secuencia de diferentes DSOs que sepan hacer uso de vinculación dinámica explícita, encadenándose al tiempo de ejecución las operaciones de perfilado de todos ellos y creando así una *suite* de herramientas apiladas (Figura 1d).

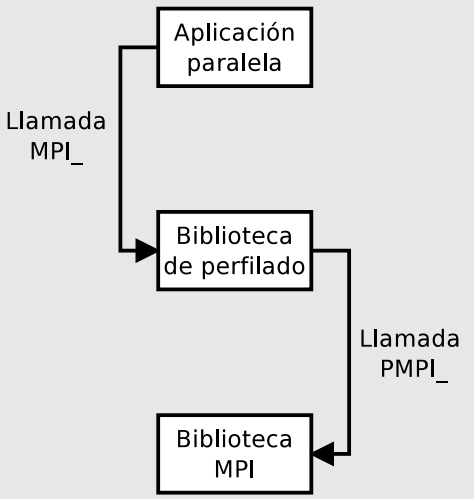
Como puede verse, el mecanismo de vinculación y carga dinámica disponible en el sistema operativo ofrece una solución para aquellas situaciones donde típicamente se necesitaría una revinculación explícita, pero no se dispone de los fuentes o de los módulos objeto independientes. Las condiciones para que una aplicación binaria pueda ser instrumentada en esta forma son que el binario sea del tipo de vinculación dinámica, y que aquellos símbolos correspondientes a las funciones a ser instrumentadas permanezcan pendientes de vinculación para el linker (en estado UNKNOWN) al tiempo de carga. Éste es el caso para la mayoría de las aplicaciones MPI que los usuarios tienen interés en trazar o perfilar.

Un DSO de usuario como el que se describe en el caso 3, interpuesto entre aplicación y otras bibliotecas, es llamado con frecuencia un *shim*, y es la técnica que emplearemos para interceptar y sustituir funciones al tiempo de ejecución en forma transparente. La técnica de instrumentación mediante un DSO interceptando llamadas a funciones puede ser llamada *de caja negra* al permitir estudiar el comportamiento y estructura de las aplicaciones, sin conocimiento de su código fuente.

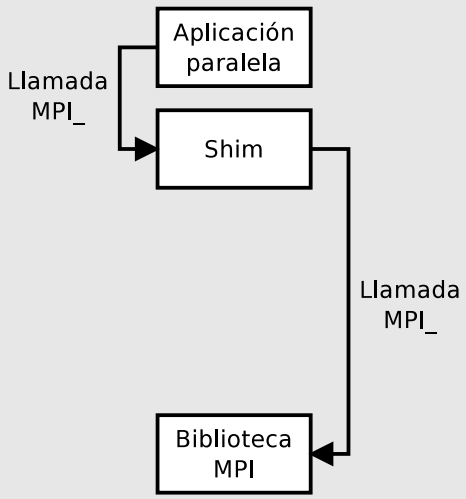
La idea de utilizar un *shim* mediante *preloading* ha sido usada para hacer instrumentación de tiempo de ejecución [44, 18]. Sin embargo, no aparecen en la literatura las técnicas de vinculación dinámica consideradas como alternativa a la instrumentación de tiempo de vinculación; es decir, aquellas herramientas que utilizan MPI Profiling Interface Layer asumen que el usuario controla el proceso de compilación y tiene capacidad de revincular la aplicación, y no mencionan la posibilidad de aplicar el caso de uso 1 recién mencionado.

4. Framework de instrumentación

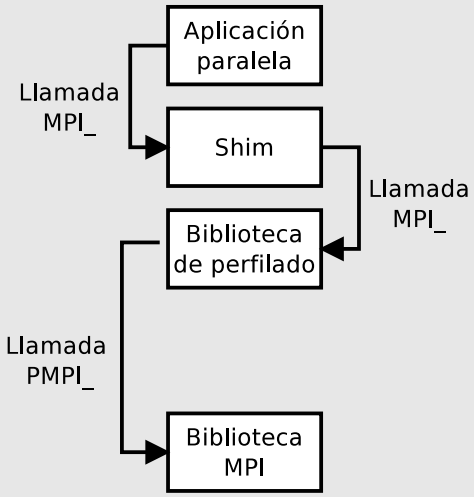
El framework de instrumentación que diseñamos aplica el mecanismo de interposición ya visto, combinando las técnicas de preloading y vinculación dinámica explícita.



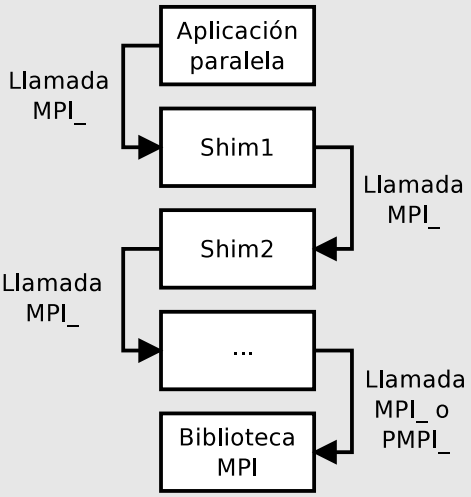
(a) Caso 1: Perfilado mediante vinculación dinámica.



(b) Caso 2: Perfilado de aplicaciones cerradas con biblioteca del usuario.



(c) Caso 3: Perfilado mixto de bibliotecas propias y de terceros usando MPI Profiling Interface Layer.



(d) Caso 4: Apilamiento de herramientas de perfilado.

Figura 1: Casos de uso para la interposición de DSOs.

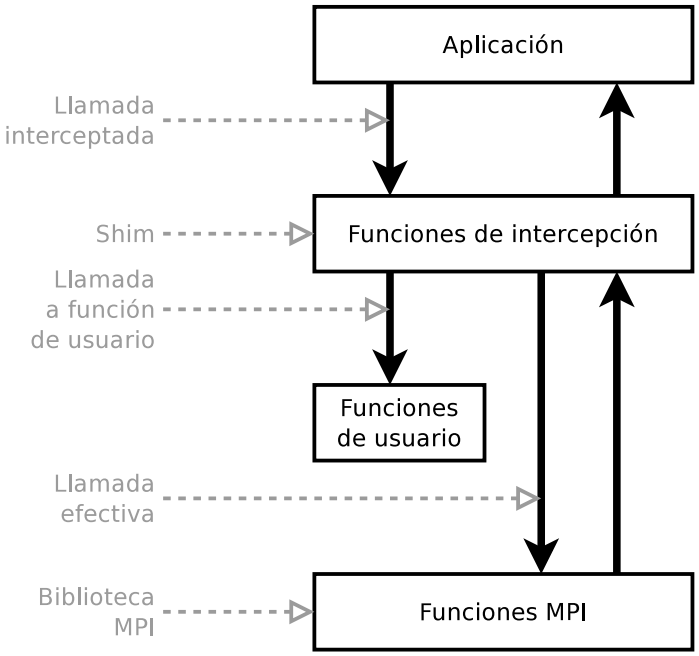


Figura 2: Arquitectura del framework de instrumentación.

4.1. Arquitectura general del framework

La arquitectura general del framework se muestra en la figura 2. Al interceptar las llamadas a funciones de biblioteca MPI, el framework es capaz de capturar toda la información relativa a los eventos de comunicaciones de la aplicación.

Nomenclatura

Para mayor claridad al explicar el diseño adoptado, y al hablar de las diferentes instancias de funciones, llamadas o invocaciones, les daremos los siguientes nombres.

Shim Una biblioteca de carga dinámica (DSO) conteniendo funciones de intercepción, que llaman a funciones de usuario y efectúan llamadas a funciones MPI.

Función de intercepción Es toda función contenida en el *shim*, cargado por *preloading*, que intercepta el control durante la ejecución de la aplicación instrumentada.

Llamada interceptada Es la invocación de una rutina MPI que hace la aplicación, y que en virtud de la interposición efectuada, transfiere el control a las funciones de intercepción que componen el *shim*.

Función de usuario Es toda función escrita por el usuario respetando una convención de llamada preestablecida y que es vinculada estática o dinámicamente al tiempo de compilación del *shim*.

Llamada a función de usuario Es la invocación de una función de usuario que hace el *shim* ante un evento MPI.

Llamada efectiva Es la invocación hecha por el *shim* de la función MPI originalmente solicitada por la aplicación.

Función MPI Es toda función de la biblioteca MPI.

4.2. Diseño del framework

Algunas decisiones de diseño adoptadas son las siguientes.

- Para permitir la mayor flexibilidad posible en las aplicaciones, adoptamos un diseño en capas y separamos arquitectónicamente la funcionalidad de instrumentación de las capacidades provistas por el usuario.
- Una pieza de software, el *shim* propiamente dicho, publica símbolos de **funciones de intercepción**, de igual nombre que las **funciones MPI**, para que sean consumidos por el linker durante el proceso de carga dinámica. Estas funciones invocarán a otras, de usuario, según un protocolo bien definido, y además se ocuparán de cargar dinámicamente la biblioteca MPI e invocar sus funciones.
- Las funciones del usuario se sitúan en relación de *callbacks* con respecto a las contenidas en el *shim*, es decir, el *shim* las invoca al recibir la **llamada interceptada**.
- Las funciones del usuario son las responsables de la creación de los registros de eventos, es decir, de la escritura de la traza. Estas funciones no necesitan ningún conocimiento de la técnica de interposición y sólo deben respetar el protocolo definido por el *shim*.
- Al interceptar cada llamada de la aplicación, el *shim* invoca a las funciones de usuario en dos momentos: antes de realizar la **llamada efectiva** hacia la capa inferior, y al regreso de ésta, ocurriendo el diagrama de interacciones que se ve en la Figura 3.
- Al realizar la **llamada a las funciones de usuario**, el *shim* les entrega toda la información que la aplicación ofreció a la función de biblioteca MPI en la **llamada interceptada**, original. Además, en la llamada a función de

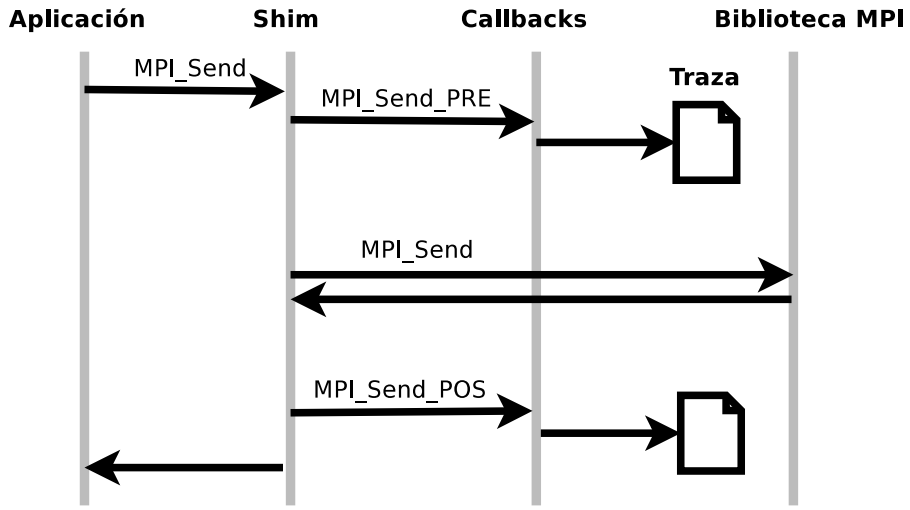


Figura 3: Diagrama de interacción, ejemplo.

usuario posterior a la **llamada efectiva**, el *shim* agrega como dato el resultado de la llamada efectiva. Por este motivo cada función de usuario lleva una signatura con cantidad y tipos de argumentos establecidos por el *shim*.

- Las funciones de usuario invocadas previamente a la **llamada efectiva** MPI llevan el sufijo *_PRE*; las funciones invocadas con posterioridad a la llamada efectiva tienen nombres que llevan el sufijo *_POS*.

- La decisión de definir **dos** funciones *callback* e invocarlas antes y después de la **llamada efectiva** permite diferentes estrategias de creación de trazas. Por ejemplo, las funciones *_PRE* son el punto donde puede registrarse información sobre la ráfaga de CPU anterior, obtenida mediante una instrumentación de tipo diferente (como las herramientas de instrumentación de nivel de máquina descriptas en 2.4). Un uso posible (típico en las aplicaciones de análisis de prestaciones) de las funciones de usuario es tomar observaciones de tiempos en ambas *callbacks* y determinar así la duración de la llamada. Las funciones *_POS* son el lugar donde ya se cuenta con la información de la duración del evento de llamada MPI, y allí se puede emitir el registro de este evento hacia la traza.

- Los lenguajes más frecuentemente usados con MPI son Fortran y C. Aun cuando el estándar MPI es único, las funciones de las bibliotecas MPI para uno y otro lenguaje tienen diferentes nombres, acordes a las convenciones usuales de uno y otro compilador. Esto ha obligado a que el *shim* contenga dos funciones de intercepción por cada función MPI, una para aplicaciones escritas en Fortran y otra para aquellas escritas en C. En cualquiera de los

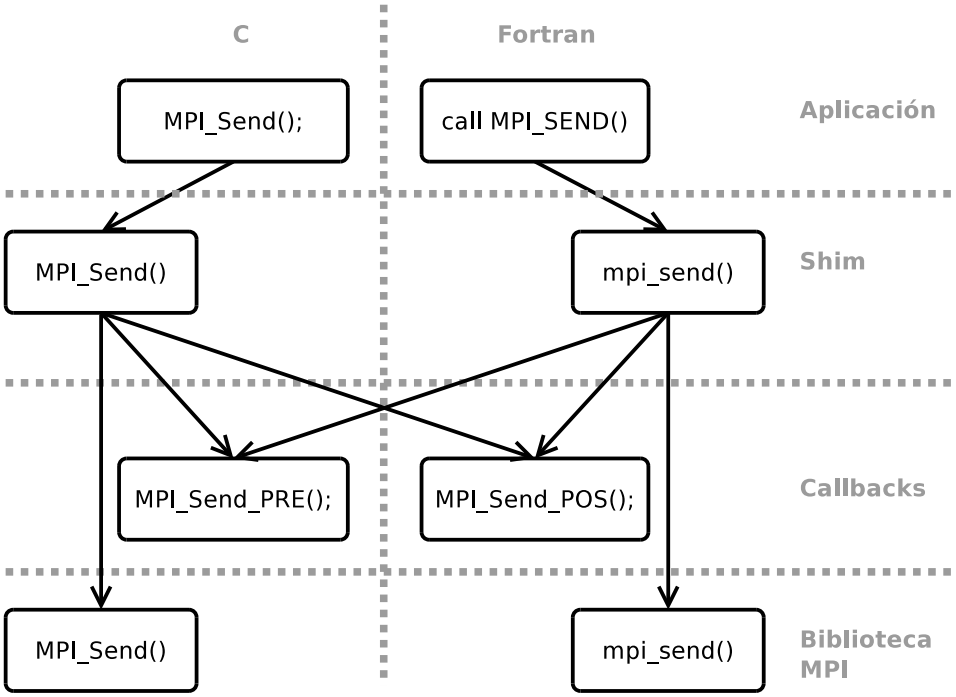


Figura 4: Nombres de funciones en el framework, ejemplo.

dos casos, existen únicas funciones *callback* de usuario *_PRE* y *_POS* para atender cada evento MPI dado (Figura 4).

Investigación del contexto

La información estadística acerca de cuántas llamadas a una función MPI dada son efectuadas durante una ejecución, con sus atributos (duración, cantidad de datos transferidos, etc.), es útil para caracterizar un programa. Normalmente, sin embargo, durante la ejecución tienen lugar un número de llamadas a una misma función MPI en diferentes puntos del programa. Conocer el comportamiento del programa en cada uno de estos puntos puede aportar información de otro orden; por ejemplo, delimitando regiones espaciales o temporales del programa donde se exhibe un perfil distintivo. Para esto es necesario poder diferenciar las llamadas según el punto del programa donde ocurren y hacer constar alguna identificación de este punto en la traza obtenida.

Este objetivo, que no es fácil cuando la instrumentación se efectúa a nivel de binarios, puede lograrse, bajo la arquitectura propuesta, por inspección del contexto. Como las funciones de intercepción forman parte del mapa de direcciones virtuales del proceso que está siendo instrumentado, tienen acceso a la pila del programa. Investigando el contexto almacenado en la pila, que contiene la direc-

```
__SHIM__CALLER __SHIM__get_caller() {
    unw_cursor_t cursor; unw_word_t sp; unw_word_t ip; unw_context_t uc;
    unw_getcontext(&uc);
    unw_init_local(&cursor, &uc);
    if((unw_step(&cursor) > 0) && (unw_step(&cursor) > 0)) {
        unw_get_reg(&cursor, UNW_REG_IP, &ip);
        return ip - CALLER0;
    } else {
        perror("caller");
        exit(1);
    }
}
```

Figura 5: Cómputo del *callsite* para una operación MPI.

ción de retorno apilada cuando la aplicación efectuó la llamada a la función MPI, una función de intercepción puede obtener dicha dirección y comunicarla a la función de usuario. Cada dirección de retorno así obtenida identifica unívocamente a un punto del programa donde se efectuó una llamada MPI.

En el framework desarrollado, esta tarea se ve facilitada por el uso de la API de la biblioteca LIBUNWIND [5]. El objetivo de esta biblioteca es definir una interfaz de programación eficiente para determinar y manipular al tiempo de ejecución la cadena dinámica de llamadas de un programa. La biblioteca resulta útil para la construcción de debuggers, para la implementación de rutinas de atención de interrupciones, o para varios fines de análisis de prestaciones.

En el vocabulario de libunwind, el punto del programa identificado por la dirección de retorno apilada recibe el nombre de *callsite*. Se ha incorporado entonces al *shim* una función llamada *__SHIM__get_caller()*, que hace uso de la función *unw_get_reg()* de la API de libunwind, para obtener la dirección del callsite asociado con la llamada MPI específica que ha sido interceptada. Esta dirección se normaliza con respecto al callsite obtenido para la llamada a *MPI_Init()* que ha sido hecha al comienzo de las operaciones (Figura 5). El tipo de dato *__SHIM__CALLER* es equivalente al de un puntero en C.

Funciones de intercepción

Diseccionaremos una función tipo del *shim* cuyo ejemplo se ve en la figura 6.

- La signatura de esta función en C es igual a la de la función MPI que intercepta.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm) {
     __SHIM__REGISTER(Send);
     caller = __SHIM__get_caller();
     MPI_Send_PRE(buf, count, datatype, dest, tag, comm, caller);
     err = __SHIM__FUNC(buf, count, datatype, dest, tag, comm);
     MPI_Send_POS(buf, count, datatype, dest, tag, comm, caller, err);
     return err;
}
```

Figura 6: Ejemplo de función del *shim* para una primitiva MPI.

- La macro *__SHIM__REGISTER()* condensa varias tareas de preparación de la llamada a la función de usuario. En particular mantiene una tabla que funciona como *cache* de punteros a funciones MPI, y prepara una variable *__SHIM__FUNC*, que apunta a la función MPI que corresponde a esta función del *shim*.
- El dato *caller* es la dirección del *callsite* obtenida por la función *__SHIM__get_caller()* y es computado durante la ejecución de la macro *__SHIM__REGISTER()*.
- Los mismos parámetros recibidos desde la aplicación por las funciones de intercepción se replican en las llamadas a las funciones de usuario, con el agregado de dos datos: *caller* (en *MPI_X_PRE()* y *MPI_X_POS()*) y *err* (en la llamada a *MPI_X_POS()*).
- El dato *err* es el resultado devuelto por la llamada a la función MPI invocada.

El caso de FORTRAN

El *shim* implementa versiones C y FORTRAN de las funciones de intercepción. Para preservar al usuario de tener que determinar si el binario proviene de C o de FORTRAN, y permitirle que diseñe y mantenga una única versión (en C) de sus funciones *callback*, adaptamos los datos recibidos de la aplicación FORTRAN a los tipos de datos del protocolo en C. A este fin utilizamos las funciones de conversión incluidas en la plataforma MPI, *MPI_Type_f2c()* y *MPI_Comm_f2c()*. En la Figura 7 se ve un ejemplo de función del *shim* para instrumentar código FORTRAN.

```
void mpi_send(char *buf, int *count, MPI_Fint *datatype, int *dest, int *tag,
MPI_Fint *comm, int *ierr) {
    __SHIM__REGISTER_F(send);
    MPI_Datatype cb_datatype; MPI_Comm cb_comm;
    caller = __SHIM__get_caller();
    cb_datatype = MPI_Type_f2c(*datatype);
    cb_comm = MPI_Comm_f2c(*comm);
    MPI_Send_PRE(buf, *count, cb_datatype, *dest, *tag, cb_comm, caller);
    __SHIM__FUNC_F(buf, count, datatype, dest, tag, comm, ierr);
    MPI_Send_POS(buf, *count, cb_datatype, *dest, *tag, cb_comm, caller, *ierr);
    return;
}
```

Figura 7: Ejemplo de función del *shim* para una primitiva MPI, en el caso de Fortran.

Timers

El *framework* permite la obtención de datos de cronometría a cargo del usuario. No existen restricciones a la cantidad de timers ni a la forma de implementación. En las experiencias de desarrollo hemos utilizado *wrappers* sobre la función POSIX *gettimeofday()*. Esta función ofrece una resolución de 1µs. Los *wrappers* mantienen la última observación recogida en una variable *clock* global, y por lo tanto visible a todas las funciones de usuario para un proceso, pero privada para dicho proceso. Una de las funciones *wrapper*, *get_elapsed(),* devuelve la diferencia del *clock* actual con respecto a la variable *clock* global.

Generación de salida

El *framework* puede ser utilizado para una cantidad de fines; típicamente, la generación de registros de información para análisis de prestaciones. Sin embargo, son posibles otros casos de uso, tales como notificar a otros procesos, locales o remotos, de condiciones que aparezcan durante la ejecución.

En el caso de uso de recolección de información, el conjunto de funciones *callback* puede diseñarse básicamente en dos maneras, de acuerdo a los fines de la instrumentación. Cada función puede ocuparse de emitir registros de eventos (para generar una traza), o bien de únicamente acumular datos escalares, tales como cantidad de llamadas, o cantidad de datos transferidos (para generar un perfil o resumen de la ejecución). En el segundo caso, el lugar más conveniente para la emisión final del resumen es en alguna de las funciones de usuario correspondiente

a *MPI_Finalize()*.

Una tercera posibilidad aparece al monitorizar eventos discretos especiales cualesquiera, que disparen la escritura de registros de resumen parcial. La aplicación de este caso de uso surge naturalmente al considerar un compromiso entre la creación de trazas (que pueden tener gran volumen y ser de difícil manejo) y de perfiles o resúmenes (que pueden aportar menos detalle del requerido).

La generación de una u otra salida queda a cargo de las funciones de usuario en la forma en que se decida implementarla. Las posibilidades inmediatas son la impresión a salida standard, la escritura de archivos secuenciales, o la emisión de las líneas de registro hacia un servidor especial sobre la red. La decisión del usuario podrá orientarse, por ejemplo, por la opción que presente menor sobrecarga en su ambiente. Los datos que figurarán en la salida de una traza podrán ser aquellos recibidos de la función de intercepción, más cualesquiera que sean computados por la función de usuario.

En la Figura 8 se ve una forma de utilización de los datos ofrecidos por el *shim* que podemos diseccionar como sigue.

- El tamaño *datasize* de los datos transferidos se calcula de acuerdo a los argumentos *datatype* y *count*.
- El identificador de comunicador MPI sirve para obtener la identificación del proceso origen (*rank*) y la cantidad de procesos en el comunicador (*size*).
- El tamaño de datos calculado anteriormente se acumula en una matriz de tráfico indexada por operación MPI.
- La función *get_elapsed()* computa el tiempo relativo, o tiempo transcurrido desde el último evento (posiblemente la llamada a la función de usuario *_POS* del evento MPI anterior).
- Finalmente se emite a salida standard un registro comprendiendo el tiempo relativo del evento, *callsite*, *rank* o número de proceso MPI que origina el evento, *rank* o número de proceso MPI que recibe los datos, cantidad y tamaño de elementos transferidos.

5. Análisis de trazas

Dado el estadio inicial de nuestro proyecto de investigación, resulta de especial interés obtener modelos descriptivos, de alto nivel, de las aplicaciones. Idealmente estos modelos deben prestarse a la interpretación humana, permitiendo el razonamiento y el aprendizaje sobre la construcción de los programas y las causas de su

```
void MPI_Send_PRE (char *buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm, __SHIM__CALLER caller) {
    int rank, size; int datasize;
    MPI_Type_size(datatype, &datasize);
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);
    sendbytes[rank][dest][Send] += count * datasize;
    printf("%lu MPI_Send_PRE caller=%u rank=%d dest=%d count=%d size=%d\n",
        get_elapsed(), caller, rank, dest, count, datasize);
    return;
}
```

Figura 8: Ejemplo de función *callback* de usuario

comportamiento, satisfactorio o no. Estos objetivos sugieren la necesidad de abordar alguna forma de *ingeniería inversa* de los programas, a partir de la impronta dejada por su ejecución.

Con el fin de explotar la información contenida en las trazas recogidas, una vez definida la metodología de instrumentación nos proponemos agregarle valor construyendo un dispositivo de análisis de trazas. El análisis de la traza tiene la misión de hallar la estructura, correspondiente a propiedades algorítmicas del programa, que subyace en la presentación secuencial de los eventos. Mientras que el comportamiento dinámico de la ejecución está contenido en una traza, su contenido es de difícil análisis. Sin embargo, un postprocesamiento adecuado de esta información puede hallar estructura en la traza y recuperar parte de la información estática que normalmente es visible sólo por medio de los fuentes.

Se abren así múltiples posibilidades:

- Obtener algún conocimiento sobre las decisiones de diseño hechas al momento de la programación, especialmente en lo relativo a las fases tempranas de *partición* y *comunicación* [25]. Investigar la naturaleza algorítmica del programa y su influencia sobre las prestaciones en un sistema paralelo dado. Identificar patrones de comunicación.

- Identificar patrones en las estrategias de resolución de problemas. Trazar clases de equivalencia entre conjuntos de programas. Es concebible que muchos programas construidos sobre una misma teoría para un dominio dado implementarán alguna variación del mismo algoritmo; si una metodología puede identificar patrones de similaridad entre programas, entonces varias cuestiones acerca de performance, escalabilidad o eficiencia pueden ser

tratadas al nivel de clases de problemas.

- Identificar problemas en el balanceo de carga y sugerir remapeos, o relocalizar dinámicamente procesos en forma eficiente. Estudiar simetría en los componentes de la aplicación. El flujo de eventos de comunicaciones puede decir si los programas tienen estructura SPMD pura o componentes con roles asimétricos, lo que puede ser usado para inferir mejores mapeos.

- Ayudar a predecir prestaciones en sistemas propuestos, reales o hipotéticos. Seleccionar hardware apropiado para correr las aplicaciones de los usuarios. Colaborar en el análisis de costo/beneficio de inversiones de acuerdo a los niveles de *speedup* esperados.

5.1. Estrategia de análisis

El framework de instrumentación se presta a la recolección de dos clases de registros: los que describen las ráfagas de cómputo (observables mediante funciones de usuario *_PRE*) y los correspondientes a eventos de comunicaciones (funciones *_POS*). Las trazas compuestas por estos registros son transformadas para construir una vista comprimida de la historia de la ejecución. Esta vista comprimida reflejará tanto como sea posible las estructuras de código fuente del programa.

Formalismo de las trazas

- Las trazas se consideran cadenas formadas por símbolos que representan a los eventos registrados.

- La compresión realizada sobre una traza puede verse como la búsqueda de una expresión regular que describe esa cadena.

- Los símbolos de la cadena se expresan como tuplas de datos, que son representantes de los registros, emitidos como se vio en 4.2.

Operador de comparación

- La estrategia de compresión operará básicamente editando la traza, reemplazando cada subsucesión de eventos esencialmente iguales (en el sentido de un operador de comparación especial) por un representante de esa subsucesión.

- Ya que las repeticiones de eventos de comunicaciones provenientes del mismo *callsite* deben poder ser atribuidas a construcciones iterativas o ciclos en

Paso	Fase	Cadena	Literales
0.	Cadena original	$AAACABABCDCDCDBB$	16
1.	1	$A[3]CABABCDCDCDB[2]$	13
2.	2	$A[3]CABAB(CD)(CD)(CD)B[2]$	10
3.	1	$A[3]CABAB(CD)[3]B[2]$	8
4.	2	$A[3]C(AB)(AB)(CD)[3]B[2]$	6
5.	1	$A[3]C(AB)[2](CD)[3]B[2]$	5

Figura 9: Ejemplo de operación del algoritmo de compresión.

los fuentes originales, el predicado de igualdad debe tener en cuenta el dato relativo al *callsite* para garantizar que los símbolos se refieren a registros generados por la misma llamada MPI.

- Por motivos similares, algunos atributos de significación local al evento registrado (tales como su tiempo de duración) serán abstraídos por el operador de comparación.

5.2. Algoritmo de compresión

El algoritmo de compresión opera sobre una cadena, en pasadas que se iteran hasta que no ocurren más cambios en una pasada. Cada pasada está compuesta por dos fases.

1. La primera fase ejecuta una búsqueda de *corridas* (sucesiones de símbolos idénticos A adyacentes) y sustituye cada corrida de n símbolos, $n > 1$, por un nuevo símbolo sintético $A[n]$.

2. La segunda fase busca el digrama maximalmente frecuente (el par (A,B) más frecuente de símbolos consecutivos) y sustituye cada ocurrencia del par en la cadena por un nuevo símbolo sintético rotulado AB.

El algoritmo de compresión se inspira en estrategias formales existentes, que involucran inferencia gramatical o de autómatas, como el algoritmo SEQUITUR [45, 43]. En la Figura 9 se muestra su operación sobre una cadena ejemplo. Como puede verse, en cada pasada se reduce efectivamente la cantidad de literales de la cadena.

Complejidad subcuadrática

Sea n la longitud original de la cadena. La complejidad de cada una de las fases es $O(n)$; luego, una pasada de dos fases tiene complejidad $O(n)$. Por la definición de las fases, cada pasada reduce la longitud de la cadena al menos en 1, o bien, el algoritmo termina. Por este motivo la cantidad de iteraciones o pasadas no puede exceder el tamaño de la entrada n, de manera que el orden de complejidad del algoritmo queda acotado superiormente por n^2. En el peor caso, se reducirá la longitud de la cadena en sólo un elemento por pasada, sin terminar hasta agotar la cadena. Habrá que efectuar así $n-1$ pasadas. El orden de este peor caso es $O(n(n-1)) = O(n^2-n)$.

5.3. Resultado del análisis

Dado el funcionamiento de la fase de búsqueda de corridas y del operador de comparación, las repeticiones de símbolos detectadas en la traza corresponden a segmentos del programa contenidos en construcciones de programación iterativas. Gracias a la fase de reconocimiento de digramas, que produce elementos sintéticos frecuentes cuya longitud se va incrementando en cada pasada, el algoritmo es capaz de reconocer las estructuras repetitivas de cualquier nivel.

Aunque dos elementos de la traza sean considerados iguales por el operador de comparación, en general poseerán atributos diferentes que pueden ser de interés para análisis a posteriori. A fin de conservar la totalidad de la información en la cadena original, cada subsecuencia de elementos reconocidos en una fase (ya sean los elementos que forman la corrida, o el par de elementos del digrama) se agrega como nodo hijo al nuevo símbolo sintético.

La traza postprocesada finalmente tiene la estructura de un árbol, construido en forma *bottom-up*, cuyos nodos internos son representantes sintéticos de corridas y digramas, y sus hojas los eventos originales en la traza. Cada nodo interno contiene información de comportamiento, resumida, sobre la subsecuencia de la cual es representante (por ejemplo, totales de tiempo insumido y de bytes transferidos por la subsecuencia de eventos).

El árbol puede ser recorrido para su examen efectuando poda a un nivel intermedio cualquiera, gracias al resumen de información o microperfil contenido en los nodos internos. Así la información contenida en la traza puede prestarse a la exploración interactiva con acción de *zoom* selectivo por niveles o ramas del árbol de análisis.

Ejemplo de análisis

En las Figuras 10 a 14 se muestra, para varios niveles de detalle, el resultado de la aplicación del algoritmo de compresión de trazas a un programa que efectúa un cómputo de la Transformada de Fourier. El programa forma parte del benchmark NAS NPB que se describirá más adelante. Explicaremos la notación adoptada.

- Numeramos las líneas de la salida para facilitar la explicación.
- Cada nodo del árbol de la traza postprocesada es de tipo *N* (nodo hoja), *D* (nodo interno, digrama), o *R* (nodo interno, corrida). Para las corridas se indica la multiplicidad entre corchetes.
- Cada par de datos numéricos $A : B$ entre paréntesis o corchetes corresponde a datos de bytes transferidos y tiempo insumido, en ese orden.
- Para cada nodo se dispone de un perfil, que es una lista de componentes que aparecen entre corchetes. El perfil resume los totales de bytes transferidos y microsegundos insumidos por cada tipo de operación, permitiendo conocer de un golpe de vista qué operaciones son responsables de la mayor transferencia de información, o en qué clase de operaciones se ha invertido mayor cantidad de tiempo. El perfil, sin embargo, no indica en qué orden ni en qué número han ocurrido estas operaciones.
- Los registros hijos de cada nodo, visibles a niveles de detalle mayores, desagregan los totales para el nodo padre tal como lo hace el perfil, pero además relatan la secuencia de operaciones en el orden y número en que han ocurrido.
- Los datos de transferencias están en bytes y los tiempos en μs. Aunque la salida normal del programa es en unidades, por claridad en las figuras, los datos mayores que 10^6 se abrevian como múltiplos de M. De esta forma, una transferencia de 1M corresponde a un megabyte, y un tiempo indicado como 1M corresponde a un millón de μs, o sea, 1s.

Nivel de detalle 0 (Figura 10). La traza completa se resume en un único nodo que es un *digrama* (registro con prefijo *D*). Los nodos hijos del raíz son invisibles debido al nivel de detalle, y totalizan 67 MB transferidos y 5s (5M μs) empleados.

En el perfil del nodo se registra una cantidad de invocaciones a la primitiva colectiva MPI *Alltoall* que totalizan 67 MB transferidos y han empleado 829050 μs ($[Alltoall : 67M : 829050]$). Similarmente, los restantes componentes del perfil del nodo son una cantidad de invocaciones a las primitivas *Barrier*, *Bcast* y *Reduce*, más ráfagas de CPU, que totalizan la cantidad de bytes transferidos y los

```
1 D (67M:5M):[Alltoall:67M:829050][Barrier:0:1850][Bcast:20:379][CPU:0:4M][Reduce:96:18429]
```

Figura 10: Programa FT, nivel de detalle 0.

```
1 D (67M:5M):[Alltoall:67M:829050][Barrier:0:1850][Bcast:20:379][CPU:0:4M][Reduce:96:18429]
2  D (67M:4M):[Alltoall:67M:829050][Barrier:0:1850][Bcast:20:379][CPU:0:3M][Reduce:80:16165]
3  D (16:245845):[CPU:0:243581][Reduce:16:2264]
```

Figura 11: Programa FT, nivel de detalle 1.

tiempos que se indican en cada caso ($[Barrier : 0 : 1850][Bcast : 20 : 379][CPU : 0 : 4M][Reduce : 96 : 18429]$).

De la salida para este primer nivel de detalle ya se puede apreciar que la mayor parte del tiempo de ejecución, alrededor de un 80 % (4s de los 5s totales), se invierte en cómputo. El componente inmediatamente siguiente en utilización de tiempo es un conjunto de llamadas a *Alltoall*, que además es la principal responsable de la actividad de E/S.

Nivel de detalle 1 (Figura 11). Al nivel 1 de detalle el digrama raíz se ve compuesto por otros dos, lo cual es indicado por la indentación. Los digramas hijos aparecen en orden según la traza. El primer digrama hijo (línea 2) es responsable por la mayor parte del tiempo insumido, que se concentra en cómputo y en demoras por E/S de la primitiva *Alltoall*.

Nivel de detalle 2 (Figura 12). Al siguiente nivel de detalle, aparece el primer digrama hijo (línea 2) explotado en otro digrama y una corrida (registro de línea 4, con prefijo *R*) con multiplicidad 5: el programa ingresa en un ciclo que se repite cinco veces, compuesto por ráfagas de CPU y llamadas a *Alltoall* y *Reduce*. El ciclo está dominado por cómputo (2s de los 3s totales del ciclo). Este ciclo

```
1 D (67M:5M):[Alltoall:67M:829050][Barrier:0:1850][Bcast:20:379][CPU:0:4M][Reduce:96:18429]
2  D (67M:4M):[Alltoall:67M:829050][Barrier:0:1850][Bcast:20:379][CPU:0:3M][Reduce:80:16165]
3   D (25M:1M):[Alltoall:25M:312229][Barrier:0:1850][Bcast:20:379][CPU:0:1M]
4   R [5] (41M:3M):[Alltoall:41M:516821][CPU:0:2M][Reduce:80:16165]
5  D (16:245845):[CPU:0:243581][Reduce:16:2264]
6   N CPU:1004:0:1:0:243581
7   N Reduce:1004:0:1:16:2264
```

Figura 12: Programa FT, nivel de detalle 2.

```
 1 D (67M:5M):[Alltoall:67M:829050][Barrier:0:1850][Bcast:20:379][CPU:0:4M][Reduce:96:18429]
 2  D (67M:4M):[Alltoall:67M:829050][Barrier:0:1850][Bcast:20:379][CPU:0:3M][Reduce:80:16165]
 3   D (25M:1M):[Alltoall:25M:312229][Barrier:0:1850][Bcast:20:379][CPU:0:1M]
 4    D (8M:688766):[Alltoall:8M:103970][Barrier:0:1850][Bcast:20:379][CPU:0:582567]
 5    R [2] (16M:1M):[Alltoall:16M:208259][CPU:0:836499]
 6   R [5] (41M:3M):[Alltoall:41M:516821][CPU:0:2M][Reduce:80:16165]
 7    D (8M:611216):[Alltoall:8M:104271][CPU:0:503069][Reduce:16:3876]
 8    D (8M:610959):[Alltoall:8M:101102][CPU:0:503889][Reduce:16:5968]
 9    D (8M:613642):[Alltoall:8M:104471][CPU:0:504784][Reduce:16:4387]
10    D (8M:614213):[Alltoall:8M:104008][CPU:0:508593][Reduce:16:1612]
11    D (8M:620147):[Alltoall:8M:102969][CPU:0:516856][Reduce:16:322]
12  D (16:245845):[CPU:0:243581][Reduce:16:2264]
13   N CPU:1004:0:1:0:243581
14   N Reduce:1004:0:1:16:2264
```

Figura 13: Programa FT, nivel de detalle 3.

también contiene la mayor parte de la E/S del programa (41MB de *Alltoall* sobre los 67MB del programa total).

El segundo digrama hijo del raíz (línea 5) se descompone en dos nodos hoja (líneas 6 y 7). Los datos contenidos en un nodo hoja son *(operación, callsite, proceso destino, cantidad de unidades de datos transferidas, tamaño de cada unidad, tiempo)*. El dato compartido en ambos (*1004*) corresponde al *callsite* del cual proceden. Cuando no tienen sentido los datos (por ejemplo, cantidad de bytes transferidos cuando la llamada es *Wait*, o *Barrier*), se indican con valores arbitrarios (generalmente -1).

Nivel de detalle 3 (Figura 13). En este nivel, la corrida de cinco elementos vista en el nivel anterior (ahora en línea 6) se descompone en digramas (líneas 7 a 11) con gran regularidad en los tiempos insumidos. Aparece otra corrida de multiplicidad 2 en línea 5.

Detalle completo Aumentando progresivamente el nivel de detalle, se llega a la presentación completa de la traza en Figura 14. La secuencia de los nodos hoja (registros *N*) reproduce la traza tal cual fue ingresada al programa.

6. Trabajo experimental

El grupo de investigación en Física al cual se brinda apoyo, ante la decisión de renovar equipamiento de cómputo, emitió una consulta respecto de cuál sería la mejor plataforma para correr una aplicación de simulación de dinámica molecular. Las alternativas en ese momento eran de costos diferentes, y las preguntas giraban alrededor de cuestiones tan básicas como si era preferible, en términos de rendimiento, un equipo multiprocesador único dotado de ocho *cores*, o dos equipos de cuatro *cores* cada uno, formando un pequeño cluster.

```
D (67M:5M):[Alltoall:67M:829050][Barrier:0:1850][Bcast:20:379][CPU:0:4M][Reduce:96:18429]
 D (67M:4M):[Alltoall:67M:829050][Barrier:0:1850][Bcast:20:379][CPU:0:3M][Reduce:80:16165]
  D (25M:1M):[Alltoall:25M:312229][Barrier:0:1850][Bcast:20:379][CPU:0:1M]
   D (8M:688766):[Alltoall:8M:103970][Barrier:0:1850][Bcast:20:379][CPU:0:582567]
    D (8M:686916):[Alltoall:8M:103970][Bcast:20:379][CPU:0:582567]
     D (8M:518211):[Alltoall:8M:103970][Bcast:20:379][CPU:0:413862]
      R [5] (20:688):[Bcast:20:379][CPU:0:309]
       D (4:307):[Bcast:4:60][CPU:0:247]
        N CPU:1001:-1:1:0:247
        N Bcast:1001:-1:1:4:60
       D (4:26):[Bcast:4:8][CPU:0:18]
        N CPU:1001:-1:1:0:18
        N Bcast:1001:-1:1:4:8
       D (4:34):[Bcast:4:7][CPU:0:27]
        N CPU:1001:-1:1:0:27
        N Bcast:1001:-1:1:4:7
       D (4:41):[Bcast:4:33][CPU:0:8]
        N CPU:1001:-1:1:0:8
        N Bcast:1001:-1:1:4:33
       D (4:280):[Bcast:4:271][CPU:0:9]
        N CPU:1001:-1:1:0:9
        N Bcast:1001:-1:1:4:271
      D (8M:517523):[Alltoall:8M:103970][CPU:0:413553]
       N CPU:1002:-1:524288:0:413553
       N Alltoall:1002:-1:524288:16:103970
     N CPU:1003:-1:-1:0:168705
    N Barrier:1003:-1:-1:0:1850
   R [2] (16M:1M):[Alltoall:16M:208259][CPU:0:836499]
    D (8M:498424):[Alltoall:8M:105126][CPU:0:393298]
     N CPU:1002:-1:524288:0:393298
     N Alltoall:1002:-1:524288:16:105126
    D (8M:546334):[Alltoall:8M:103133][CPU:0:443201]
     N CPU:1002:-1:524288:0:443201
     N Alltoall:1002:-1:524288:16:103133
  R [5] (41M:3M):[Alltoall:41M:516821][CPU:0:2M][Reduce:80:16165]
   D (8M:611216):[Alltoall:8M:104271][CPU:0:503069][Reduce:16:3876]
    D (16:245343):[CPU:0:241467][Reduce:16:3876]
     N CPU:1004:0:1:0:241467
     N Reduce:1004:0:1:16:3876
    D (8M:365873):[Alltoall:8M:104271][CPU:0:261602]
     N CPU:1002:-1:524288:0:261602
     N Alltoall:1002:-1:524288:16:104271
   D (8M:610959):[Alltoall:8M:101102][CPU:0:503889][Reduce:16:5968]
    D (16:248045):[CPU:0:242077][Reduce:16:5968]
     N CPU:1004:0:1:0:242077
     N Reduce:1004:0:1:16:5968
    D (8M:362914):[Alltoall:8M:101102][CPU:0:261812]
     N CPU:1002:-1:524288:0:261812
     N Alltoall:1002:-1:524288:16:101102
   D (8M:613642):[Alltoall:8M:104471][CPU:0:504784][Reduce:16:4387]
    D (16:247382):[CPU:0:242995][Reduce:16:4387]
     N CPU:1004:0:1:0:242995
     N Reduce:1004:0:1:16:4387
    D (8M:366260):[Alltoall:8M:104471][CPU:0:261789]
     N CPU:1002:-1:524288:0:261789
     N Alltoall:1002:-1:524288:16:104471
   D (8M:614213):[Alltoall:8M:104008][CPU:0:508593][Reduce:16:1612]
    D (16:244280):[CPU:0:242668][Reduce:16:1612]
     N CPU:1004:0:1:0:242668
     N Reduce:1004:0:1:16:1612
    D (8M:369933):[Alltoall:8M:104008][CPU:0:265925]
     N CPU:1002:-1:524288:0:265925
     N Alltoall:1002:-1:524288:16:104008
   D (8M:620147):[Alltoall:8M:102969][CPU:0:516856][Reduce:16:322]
    D (16:251159):[CPU:0:250837][Reduce:16:322]
     N CPU:1004:0:1:0:250837
     N Reduce:1004:0:1:16:322
    D (8M:368988):[Alltoall:8M:102969][CPU:0:266019]
     N CPU:1002:-1:524288:0:266019
     N Alltoall:1002:-1:524288:16:102969
 D (16:245845):[CPU:0:243581][Reduce:16:2264]
  N CPU:1004:0:1:0:243581
  N Reduce:1004:0:1:16:2264
```

Figura 14: Análisis de traza para el programa FT, nivel de detalle completo.

Evidentemente, a iguales prestaciones de cómputo, la arquitectura de cuatro núcleos ofrece vínculos para la comunicación, entre cada par de procesadores, que son de prestaciones inevitablemente superiores a las de un cluster construido sobre una red multipropósito. Sin embargo, es pertinente la pregunta de si, dada una aplicación que corre en una plataforma conocida, y dada una nueva plataforma a un cierto costo, esa aplicación en particular será capaz de aprovechar la mejora en comunicaciones. Dicho de otra manera: ¿cuál podrá ser, para la aplicación del usuario, el *speedup* esperado en esta nueva plataforma?. Y, no menos importante, ¿cuánto está dispuesto a pagar el usuario por cada punto de *speedup*?. Estas preguntas sugirieron una experiencia práctica completamente básica que aportó conocimiento sobre éste y otros casos de migración a las nuevas generaciones de equipos *multicore*.

6.1. MPI sobre multicores

Dadas las tendencias actuales, a medida que la cantidad de núcleos de cómputo se incrementa, el acceso a memoria se vuelve un cuello de botella. En respuesta a este problema, los diseños más recientes de multicores dividen la memoria en bancos, los controladores de memoria se incorporan a los procesadores, y a éstos se les asignan bancos de memoria privados. Un procesador actual típico incluye una cantidad de núcleos de cómputo, junto con un área "*uncore*" de recursos comunes, tales como la interconexión entre los núcleos y controladores de memoria. El estado de evolución de estos procesadores presenta por lo tanto diseños NUMA (Non Uniform Memory Access), con diferentes distancias de acceso entre procesadores y memoria.

Los sistemas actuales se construyen sobre varios de estos procesadores. Los *interconnects* punto a punto para estos sistemas están especialmente diseñados para transportar tráfico de datos y señales de coherencia de cachés entre los procesadores, reemplazando estrategias de bus utilizadas anteriormente [36].

Al momento de ejecutar una aplicación MPI sobre un equipo de estas características, aparecen una cantidad de variables que influyen sobre el rendimiento y que el usuario puede llegar a manipular. No solamente el sistema operativo, sino también los sistemas de tiempo de ejecución MPI actuales, ofrecen mecanismos para modificar ciertos modos de acción.

Asignación de núcleos

Los kernels SMP Linux ofrecen políticas de asignación de núcleos a procesos [37]. A nivel del sistema, se dispone de parámetros de arranque para determinar el aislamiento (*isolation*) de núcleos, o bien los núcleos pueden ser activados o desactivados dinámicamente durante la vida del sistema. A nivel de procesos, éstos

pueden ser disparados con información de afinidad, para indicar al scheduler que deben ser excluidos de ciertos procesadores.

La memoria en los nodos NUMA se asigna a los threads de acuerdo a la política *first-touch*. Esto significa que el thread que primero referencia una posición de memoria logrará que dicha memoria sea tomada del pool del procesador donde ese thread corre, de modo de minimizar la distancia de acceso [23]. La regla de eficiencia complementaria es *owner-computes*, significando que el thread dueño de los datos (y por lo tanto el procesador dueño del banco de memoria) es el responsable de los cómputos sobre ese dato. El scheduler de Linux tiende a mantener un proceso en la misma CPU donde comenzó. Esto es conveniente por razones de reuso de cachés, pero especialmente se aplica a nodos NUMA, donde la migración de procesos entre procesadores es aún más costosa [35].

Parámetros de ejecución de MPI

Descripción del hardware El sistema de runtime de Open MPI permite imponer políticas de ejecución que alienten o desalienten la asignación de los procesos a ciertos núcleos. El usuario puede describir con precisión el hardware subyacente y el mapeo deseado de procesos MPI a los núcleos individuales. La noción de *slot* (identificador local al nodo que designa núcleos, o equivalentemente, threads independientes) sirve a estos fines.

Tamaño de los mensajes Algunas implementaciones de MPI permiten al usuario sintonizar los límites entre mensajes considerados *cortos* y *largos*, lo cual determina por motivos de eficiencia los diferentes protocolos con los cuales serán traficados (respectivamente *eager* y *rendez-vous*).

Byte Transfer Layer (BTL) Los runtimes de MPI sobre arquitecturas multicore normalmente son capaces de seleccionar la implementación de los mensajes. Open MPI, en particular, puede conmutar los modos de comunicación entre memoria compartida y segmentos TCP, dependiendo de si los procesos que se comunican se encuentran o no sobre el mismo nodo. El framework Byte Transfer Layer (BTL) de Open MPI provee una forma genérica de implementar drivers de bajo nivel para cada canal de comunicación [33]. Aunque la característica MCA (Modular Component Architecture) ofrece un mecanismo general para inducir uno u otro modo en forma forzada, Open MPI automáticamente explota la plataforma multicore utilizando el framework BTL.

6.2. Densidad de núcleos

Investigamos las consecuencias de variar la cantidad de núcleos por equipo (o *densidad*) en un cluster. La idea de la experiencia fue correr un conjunto variado de cargas de trabajo sobre dos escenarios de referencia. Nuestro laboratorio se montó con dos equipos de dos procesadores ("*dual-socket*") multicore. Los equipos son Intel S5500BC con ambos sockets ocupados por procesadores Intel Xeon E5502 a 1.87GHz, con 16GB RAM. La topología de estas máquinas NUMA se ve en la Figura 15 según la describe el programa *hwloc* [12]. Cada procesador tiene dos núcleos con caché L1 y L2 privadas pero caché compartida ("*uncore*") L3.

- El escenario CLUSTER representa la elección habitual del usuario que viene del trabajo con clusters de equipos unicore. Es un pequeño cluster formado por los dos equipos, en cada uno de los cuales han sido aislados (*isolated*) los dos núcleos de un procesador al momento del arranque. La configuración tiene por consiguiente cuatro núcleos pero con interconexiones de propiedades sumamente diferentes. El enlace entre núcleos de un mismo procesador tiene las propiedades de latencia y ancho de banda de un interconnect como QPI, mientras que núcleos de procesadores diferentes están en nodos diferentes del cluster y por lo tanto quedan comunicados por la red (en nuestro laboratorio, Ethernet conmutada a 1Gbps).

- El escenario MULTICORE consiste en un solo equipo de los anteriores, con los cuatro núcleos habilitados. En este escenario todos los núcleos están conectados por canales QPI.

En ambos casos se corren cuatro procesos MPI, uno sobre cada uno de los cuatro núcleos disponibles. La implementación MPI es Open MPI 1.3.2. El sistema operativo es CentOS 5.4 de 64 bits, con kernel actualizado, sin modificaciones.

Benchmark NAS NPB

Para estimar la diferencia entre las arquitecturas se eligió el conocido benchmark NPB. NAS NPB (NASA Parallel Benchmark) es un benchmark diseñado por NASA para evaluación de hardware [10]. Consiste en varios programas usados en Dinámica de Fluidos Computacional (Tabla 1). La suite es de acceso libre y comprende varias implementaciones: secuencial, paralelo de memoria compartida (OpenMP) y paralelo de paso de mensajes (MPI). La mayoría de los programas están en Fortran, algunos en C. Desde la versión 3.3 existen implementaciones en Java y en HPFortran.

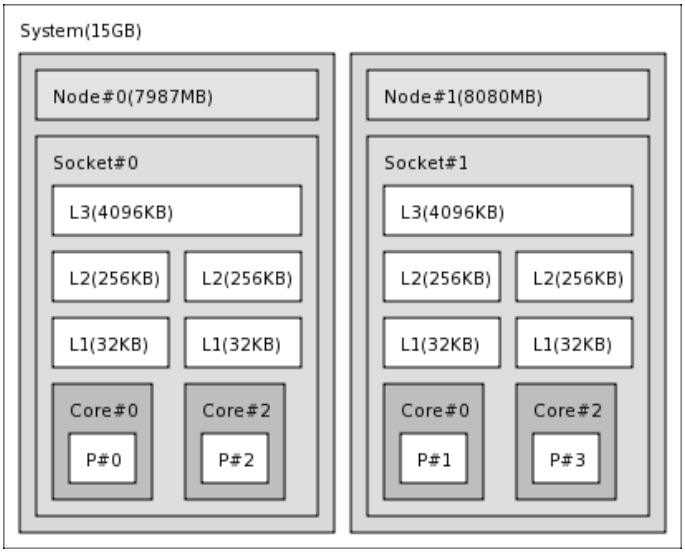


Figura 15: Arquitectura de los sistemas del laboratorio.

Los programas de la suite abordan problemas en varios tamaños de datos predefinidos, llamados clases. El tamaño exacto de problema representado por las clases es diferente para cada programa pero están idénticamente ordenados para cada programa (Tabla 2). En nuestro laboratorio, las clases D y superiores causaron *swapping*, de modo que no fueron incluidos en la experiencia. Las clases S y W fueron demasiado pequeñas para dar resultados confiables, por lo cual también fueron desestimadas. Finalmente se corrió la versión MPI de los benchmarks sobre las clases A, B y C.

Ancho de banda

Utilizamos una aplicación de *ping-pong* sencilla para observar la diferencia a priori entre los tipos de interconexión entre procesadores de ambos escenarios. La Figura 16 demuestra que Open MPI es capaz de aprovechar las mejores propiedades del canal QPI en el escenario MULTICORE. En líneas generales, mayores tamaños de mensaje ofrecen mayor ancho de banda utilizado.

La tercera curva (*multicore/cluster*) muestra la relación entre la velocidad de transferencia del canal QPI, según es aprovechada por Open MPI, y la velocidad del canal dedicado 1Gbps Ethernet. La relación entre ambas curvas está magnificada 100 veces por motivos de escala y cursa valores desde el orden de 10X (para

BT	BT es una aplicación de simulación de dinámica de fluidos que usa un algoritmo implícito para resolver ecuaciones de Navier-Stokes tridimensionales. La solución, por diferencias finitas, se basa en factorización aproximada *Alternating Direction Implicit (ADI)* que desacopla las tres dimensiones. Los sistemas resultantes son bloque-tridiagonales de 5x5 bloques y se resuelven secuencialmente sobre cada dimensión.
SP	SP es una aplicación de simulación de dinámica de fluidos que tiene una estructura similar a BT. La solución, por diferencias finitas, se basa en una factorización aproximada por método Beam-Warming que desacopla las tres dimensiones. El sistema resultante es de bandas pentadiagonales que se resuelven secuencialmente sobre cada dimensión.
LU	LU es una aplicación de simulación de dinámica de fluidos que usa el método de sobrerrelajaciones sucesivas (SSOR) para resolver un sistema de siete bloques diagonales resultante de la discretización en diferencias finitas de las ecuaciones de Navier-Stokes por división en sistemas triangulares en bloques inferior y superior.
FT	FT contiene el kernel computacional de un método espectral basado en Transformada Rápida de Fourier (FFT) en tres dimensiones. El programa FT ejecuta una FFT unidimensional por cada dimensión.
CG	CG usa un método de Gradiente Conjugado para computar una aproximación al autovalor mínimo de una matriz grande, dispersa, no estructurada. Este kernel evalúa cómputos y comunicación en malla no estructurada usando una matriz con ubicaciones de entradas aleatoriamente generadas.
EP	EP es un benchmark de tipo *Embarrassingly Parallel* (escandalosamente paralelo). Genera pares de desviaciones gaussianas de acuerdo a un esquema específico. Su meta es establecer el punto de referencia para la performance pico de una plataforma dada. El programa EP es casi independiente de la interconexión entre procesadores ya que la comunicación es mínima.
MG	MG usa un método Vcycle MultiGrid para computar la solución de la ecuación escalar de Poisson en tres dimensiones. El algoritmo trabaja en forma continua sobre un conjunto de mallas que se construyen entre gruesa y fina. Evalúa movimientos de datos sobre corta y larga distancia.
IS	IS es un algoritmo de ordenamiento (*sort*) paralelo con muy alta sensibilidad a la latencia del interconnect.

Cuadro 1: Programas de la suite NAS NPB3.3.

	BT	CG	EP	FT	IS	LU	MG	SP
S	12x12x12	1400	33554432	64x64x64	65536	12x12x12	32x32x32	12x12x12
W	24x24x24	7000	67108864	128x128x32	1048576	33x33x33	128x128x128	36x36x36
A	64x64x64	14000	536870912	256x256x128	8388608	64x64x64	256x256x256	64x64x64
B	102x102x102	75000	2147483648	512x256x256	33554432	102x102x102	256x256x256	102x102x102
C	162x162x162	150000	8589934592	512x512x512	134217728	162x162x162	512x512x512	162x162x162

Cuadro 2: Tamaños de datos para las clases del benchmark NPB3.3.

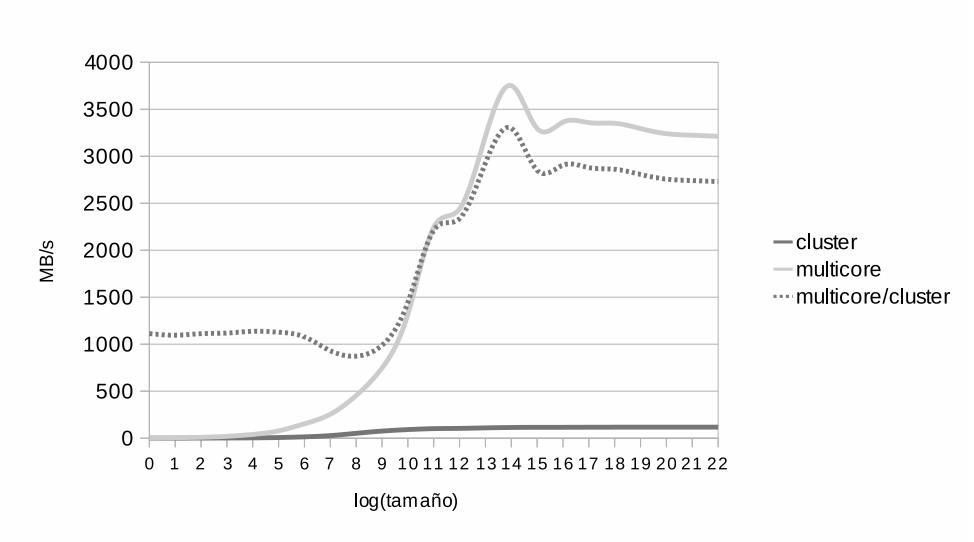


Figura 16: Ancho de banda disponible entre procesadores, en ambos escenarios.

tamaños de hasta 1KB) hasta 32X (alrededor de los 2^{14} bytes). Aunque lejos de la relación teórica, de 250X [41], las curvas confirman que Open MPI conmuta automáticamente el dispositivo a nivel de BTL para comunicar procesadores cercanos y por lo tanto es esperable mejor rendimiento, en general, en el escenario MULTICORE para aquellas aplicaciones que demanden utilización de red.

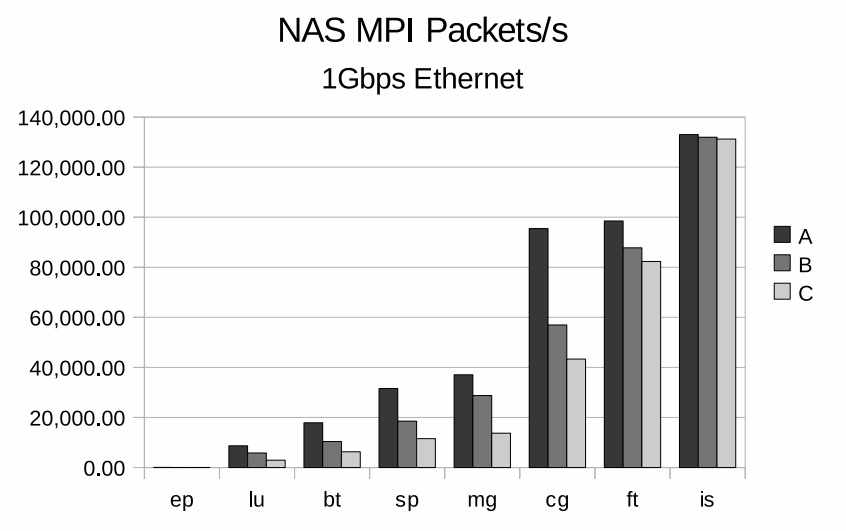
Utilización de red

Las aplicaciones del benchmark presentan diferentes perfiles de consumo de red. Por inspección de la interfaz Ethernet de los nodos en el escenario CLUSTER, obtuvimos los perfiles que se muestran en Figura 17. Los programas de la suite se muestran en el orden inducido por los valores observados.

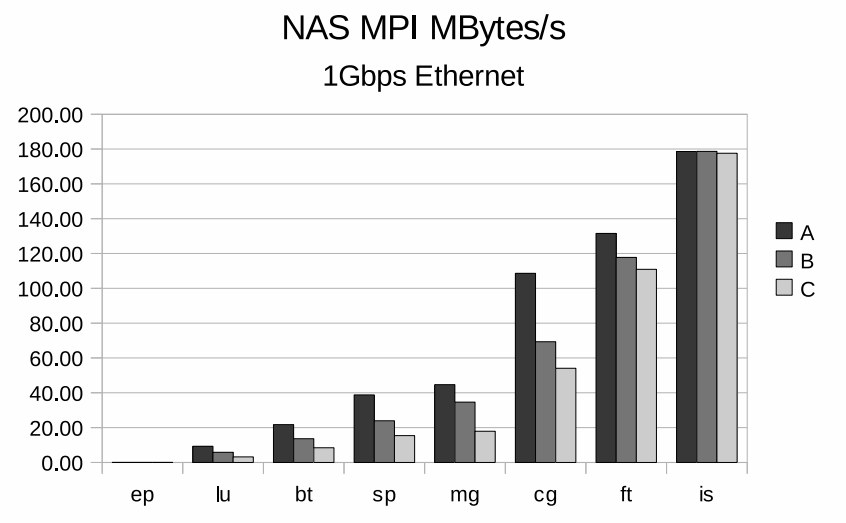
Speedups

Designamos como *speedup* de ejecución de cada programa de la suite en cada tamaño de datos, a la relación entre tiempos de ejecución en ambos escenarios:

$$s = \frac{t_{cluster}}{t_{multicore}}$$



(a) Paquetes por segundo.



(b) Megabytes por segundo.

Figura 17: Actividad de red para los programas de la suite NAS NPB3.3, tamaños A, B y C.

Esta relación define la mejora relativa en comportamiento, como consecuencia de pasar del escenario tradicional de CLUSTER, con menor densidad de núcleos por nodo, al escenario de mayor densidad (MULTICORE). Se efectuaron cien corridas de cada instancia de programa en cada clase, y se promediaron. Los datos se presentan en Figura 18. Nuevamente, los nombres de los programas aparecen en el orden inducido por los valores observados de utilización de red.

6.3. Aplicación del framework

Como se ve en la Figura 18, los mayores speedups se encuentran alrededor de 2.5. Tal como se esperaba, ninguno de los programas corre más rápidamente en el escenario CLUSTER que en MULTICORE. Sin embargo, algunos programas como EP, LU o BT obtienen speedups muy cercanos a 1, es decir, no hay ganancia considerable al pasar del escenario tradicional al de mayor densidad. El ranking de programas de la suite obtenido anteriormente, evaluando la actividad de red, coincide con el de speedups observados.

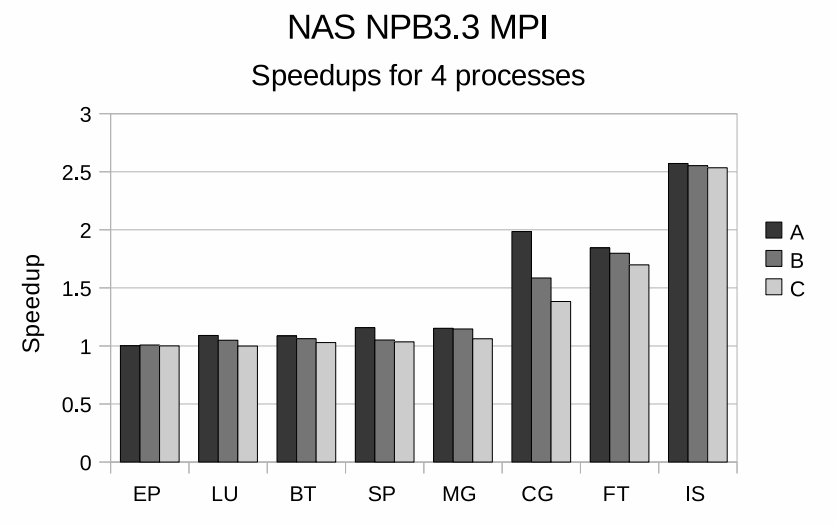
Sin embargo, el bajo aprovechamiento del canal de mayor velocidad no queda explicado por esta correlación sino por la relación entre tiempo de cómputo y de comunicaciones. Como lo predice la Ley de Amdahl, los speedups observados se explican por la naturaleza de las aplicaciones. Los programas con mayor proporción de tiempo empleado en uso de red tienen mejores chances de aprovechar la plataforma propuesta (el escenario MULTICORE). Los programas que presenten perfil similar al de las aplicaciones escandalosamente paralelas mostrarán menos impacto. Para estos últimos, el cambio de plataforma debe aportar otra propuesta de valor (menores costos monetarios, ahorros en consumo energético...) para que se justifique aceptarlo.

Utilizando el framework de instrumentación y análisis de trazas, se habría podido explorar la relación entre los programas de la suite en una sola corrida, prescindiendo de todo el trabajo experimental anterior. Por ejemplo, los programas de la suite más diferentes entre sí son EP e IS, que ocupan los extremos opuestos del espectro en todos los rankings. Este resultado pudo haberse predicho a partir del análisis de las trazas obtenidas según hemos expuesto en este trabajo. El programa EP, como muestra la Figura 19, ocupa prácticamente la totalidad del tiempo en operaciones de cómputo (7 s en CPU de los 7 s totales); mientras que IS (Figura 20) dedica sólo un 70 % del tiempo de ejecución (760 ms de 1 s) a cómputos.

La metodología desarrollada permite comenzar a contestar preguntas como las que motivaban esta experiencia, en el menor tiempo posible, al único costo de una corrida de la aplicación del usuario; sin requerir acceso a los fuentes y sin el requisito de dominar la operatoria de los algoritmos.

	A	B	C
EP	1	1.01	1
LU	1.09	1.05	1
BT	1.09	1.06	1.03
SP	1.16	1.05	1.04
MG	1.15	1.15	1.06
CG	1.99	1.59	1.38
FT	1.84	1.8	1.7
IS	2.57	2.55	2.53

(a) Speedups para el benchmark.



(b) Speedups para el benchmark, gráficamente.

Figura 18: Relación de tiempos de ejecución entre escenarios CLUSTER y MULTICORE.

```
D (108:7M):[Allreduce:104:54938][Barrier:0:141][Bcast:4:66][CPU:0:7M]
 D (4:1830):[Barrier:0:141][Bcast:4:66][CPU:0:1623]
  D (4:1689):[Bcast:4:66][CPU:0:1623]
  N Barrier:1002:-1:-1:0:141
 R [4] (104:7M):[Allreduce:104:54938][CPU:0:7M]
  D (8:7M):[Allreduce:8:54655][CPU:0:7M]
  D (8:129):[Allreduce:8:108][CPU:0:21]
  D (80:116):[Allreduce:80:66][CPU:0:50]
  D (8:122):[Allreduce:8:109][CPU:0:13]
```

Figura 19: Programa EP, nivel de detalle medio.

```
D (92M:1M):[Allreduce:45276:1595][Alltoall:44:1313][Alltoallv:92M:308784][Bcast:4:47][CPU:0:760082][Reduce:12:1584][Send:4:45]
 D (92M:1M):[Allreduce:45276:1595][Alltoall:44:1313][Alltoallv:92M:308784][Bcast:4:47][CPU:0:760082][Reduce:8:202][Send:4:45]
  D (92M:1M):[Allreduce:45276:1595][Alltoall:44:1313][Alltoallv:92M:308784][Bcast:4:47][CPU:0:755850][Reduce:8:202][Send:4:45]
  N CPU:1007:0:1:0:4232
 N Reduce:1007:0:1:4:1382
```

Figura 20: Programa IS, nivel de detalle medio.

7. Conclusiones

Se ha desarrollado una herramienta de análisis de prestaciones para aplicaciones paralelas de paso de mensajes que permite extraer conocimiento sobre la estructura de los programas, aun sin acceso a los fuentes ni a los algoritmos empleados.

La herramienta se divide en dos componentes: un módulo que funciona al tiempo de ejecución de las aplicaciones, obteniendo trazas según requerimientos del usuario, y una aplicación del módulo anterior que analiza las trazas recogidas, obteniendo conocimiento sobre la naturaleza algorítmica de las aplicaciones ejecutadas y sobre los recursos utilizados. Este conocimiento permite hacer predicciones bajo condiciones diferentes de ejecución.

Se ha mostrado una aplicación de la metodología de análisis de prestaciones que asiste en la formulación de recomendaciones preliminares respecto de las mejores plataformas para cada aplicación. La herramienta permitió respaldar cuantitativamente una recomendación, basada en la teoría, sobre la elección de una plataforma de ejecución en un caso real. Durante el desarrollo que describimos se reunió una cantidad considerable de conocimiento nuevo sobre diferentes aspectos de HPC, incluyendo problemas, arquitecturas, herramientas y técnicas, conocimiento que se incorpora al acervo del proyecto de investigación en cuyo marco trabajamos.

Quedan por desarrollar varias extensiones posibles de la metodología, en particular una caracterización fina de las primitivas de MPI en términos de latencia de las operaciones y en función del tamaño de los mensajes. Aquí la dificultad consiste en estudiar con profundidad el funcionamiento detallado de cada primitiva MPI, en especial las operaciones colectivas, para obtener modelos correctos de cada una de ellas. Dicha caracterización debe permitir modelizar las trazas comprimidas, aprovechando la capacidad de identificar las ocurrencias de cada una de las llamadas, y así predecir la latencia de secuencias de operaciones para diferentes cargas de trabajo y diferentes plataformas. Con tales modelos se podrá orientar la investigación a una predicción efectiva de prestaciones de las aplicaciones.

Referencias

[1] About OProfile. http://oprofile.sourceforge.net/about/.

[2] Cómputo de altas prestaciones. http://hpc.uncoma.edu.ar/.

[3] ELF - executable and linkable format - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Executable_and_Linkable_Format.

[4] GNU gprof - table of contents. http://www.cs.utah.edu/dept/old/ texinfo/as/gprof_toc.html.

[5] The libunwind project. http://www.nongnu.org/libunwind/.

[6] Measuring and improving application performance with PerfSuite. http://perfsuite.ncsa.uiuc.edu/ publications/LJ135/t1.html.

[7] MPE. http://www.mcs.anl.gov/research/projects/mpi/www/www4/MPE.html.

[8] MPI documents. http://www.mpi-forum.org/docs/docs.html.

[9] mpiP: lightweight, scalable MPI profiling. http://mpip.sourceforge.net/.

[10] NASA advanced supercomputing (NAS) division home page. http://www.nas.nasa.gov/.

[11] Performance analysis tools. https://computing.llnl.gov/tutorials/ performance_tools/.

[12] Portable hardware locality (hwloc) documentation: v0.9.1. http://www.open-mpi.org/projects/hwloc/doc/v0.9.1/.

[13] SourceForge.net: linux performance counters driver - project web hosting - open source software. http://perfctr.sourceforge.net/.

[14] TAU - tuning and analysis utilities -. http://www.cs.uoregon.edu/research/tau/home.php.

[15] TUD - ZIH - open trace format (OTF). http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/ software_werkzeuge_zur_unterstuetzung_von _programmierung_und_optimierung/otf.

[16] UNIX man pages : dlopen (3). http://compute.cnr.berkeley.edu/cgi-bin/man-cgi?dlopen+3.

[17] Vampir. http://www.vampir.eu/.

[18] K. J Barker, K. Davis, A. Hoisie, D. J Kerbyson, M. Lang, S. Pakin, and J. C Sancho. A performance evaluation of the nehalem quad-core processor for scientific computing. *Parallel Processing Letters*, 18(4):453–469, 2008.

[19] Anthony Chan, William Gropp, and Ewing Lusk. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Sci. Program.*, 16(2-3):155–165, April 2008.

[20] Michael Collette. High performance tools and technologies. Technical Report UCRL-TR-209289, LLNL, 2004.

[21] Arnaldo Carvalho de Melo. Performance counters on linux, the new tools. http://linuxplumbersconf.org/2009/slides/Arnaldo-Carvalho-de-Melo-perf.pdf. Linux Plumbers Conference, September, 2009.

[22] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using PAPI for hardware performance monitoring on linux systems. In *Conference on Linux Clusters: The HPC Revolution*, 2001.

[23] U. Drepper. What every programmer should know about memory. *Eklektix, Inc., Október*, 2007.

[24] Ulrich Drepper. Drepper, shared libraries. http://www.redhat.com/f/summitfiles/presentation/June2/Developer%20Tools/Drepper_Writing%20Shared%20Libraries.pdf.

[25] Ian T Foster. *Designing and building parallel programs : concepts and tools for parallel software engineering*. Addison-Wesley, Reading, Mass. [u.a.], 1995.

[26] Karl Fuerlinger, Michael Gerndt, Jack Dongarra, and Technische Universität München. On using incremental profiling for the performance analysis of shared memory parallel applications.

[27] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H Castain, David J Daniel, Richard L Graham, and Timothy S Woodall. Open MPI: goals, concept, and design of a next generation MPI implementation. *IN PROCEEDINGS, 11TH EUROPEAN PVM/MPI USERS' GROUP MEETING*, pages 97—104, 2004.

[28] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. Open MPI: a flexible high performance MPI. *IN THE 6TH ANNUAL INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING AND APPLIED MATHEMATICS*, 2005.

[29] E. Grosclaude, C. Zanellato, J. Balladini, R. del Castillo, and S. Castro. Considering core density in hybrid clusters. In *39 Jornadas Argentinas de Informática e Investigación Operativa (JAIIO), HPC*, Buenos Aires, 2010.

[30] E. Grosclaude, C. Zanellato, J. Balladini, R. del Castillo, and S. Castro. Profiling MPI applications with mixed instrumentation. In *Workshop de Procesamiento Distribuido y Paralelo - CACIC 2010: XVI Congreso Argentino de Ciencias de la Computación*, pages 132–141, Morón, 2010.

[31] Eduardo Grosclaude. Computación de altas prestaciones. In *WICC 2011*, Rosario, 2011.

[32] Eduardo Grosclaude, Claudio Zanellato, Rafael Zurita, and Rodolfo del Castillo. Tracing MPI binaries for knowledge discovery. La Plata, 2011.

[33] Torsten Hoefler, Mirko Reinhardt, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. W.: Low overhead ethernet communication for open mpi on linux clusters. 2006.

[34] Zoltán Juhász. *Distributed & Parallel Systems - Cluster & Grid computing*. Springer, 2005.

[35] V. Kazempour, A. Fedorova, and P. Alagheband. Performance implications of cache affinity on multicore processors. *Lecture Notes in Computer Science*, 5168:151–161, 2008.

[36] M. A Khan. Optimization study for multicores. *Department of IT, Uppsala University*.

[37] A. Kleen. A NUMA API for linux. *Novel Inc*, 2005.

[38] Andreas Knüpfer. OTF tools.

[39] A. Leko, H. Sherburne, H. Su, B. Golden, and A. D George. *Practical experiences with modern parallel performance analysis tools: An evaluation*. Citeseer.

[40] Emilio Luque and Tomàs Margalef. *Euro-Par 2008 - parallel processing: 14th International Euro-Par Conference, Las Palmas de Gran Canaria, Spain, August 26-29, 2008 ; proceedings*. Springer, 2008.

[41] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 261–270, Raleigh, North Carolina, USA, September 2009.

[42] Shirley Moore, David Cronk, Kevin London, and Jack Dongarra. Review of performance analysis tools for MPI parallel programs. *IN Y. COTRONIS AND J. DONGARRA, EDS., RECENT ADVANCES IN PARALLEL VIRTUAL MACHINE AND MESSAGE PASSING INTERFACE, 8TH EUROPEAN PVM/MPI USERS' GROUP MEETING, PROCEEDINGS, LNCS 2131*, 241, 1998.

[43] C. G Nevill-Manning. *Inferring sequential structure*. PhD thesis, Citeseer, 1996.

[44] R. Rabenseifner. Automatic profiling of MPI applications with hardware performance counters. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, page 22–22, 1999.

[45] David Salomon. *Data Compression The Complete Reference - 4th Ed - 2007*.

[46] J. S Vetter and M. O McCracken. Statistical scalability analysis of communication operations in distributed applications. *ACM SIGPLAN Notices*, 36(7):123–132, 2001.